

Contents

COMPILER DESIGN (VI-SEM., CS-BRANCH)

	PAGE NO.
UNIT-I: INTRODUCTION TO COMPILING & LEXICAL ANALYSIS	
Introduction of Compiler, Major data Structure in compiler, Types of compiler, Front-end and Back-end of Compiler.....	(03 to 13)
Compiler structure – Analysis-synthesis model of compilation, various phases of a compiler	(13 to 21)
Lexical analysis – Input buffering, Specification & Recognition of Tokens, Design of a Lexical Analyser Generator, LEX	(21 to 42)
UNIT-II : SYNTAX ANALYSIS & SYNTAX DIRECTED TRANSLATION	
Syntax analysis – CFGs, Top down parsing, Brute force approach, recursive descent parsing.....	(43 to 55)
Transformation on the grammars, predictive parsing, bottom up parsing	(55 to 72)
Operator precedence parsing	(72 to 80)
LR parsers (SLR, LALR, LR), Parser generation.....	(80 to 110)
Syntax directed definitions – Construction of Syntax trees, Bottom up evaluation of S-attributed definition, L-attribute definition, Top down translation, Bottom Up evaluation of inherited attributes, Recursive Evaluation, Analysis of Syntax directed definition	(110 to 128)
UNIT-III : TYPE CHECKING & RUN TIME ENVIRONMENT	
Type checking – Type system, specification of simple type checker, equivalence of type expression, type conversion, overloading of functions and operations, polymorphic functions	(129 to 140)
Run time Environment – Storage organization, Storage allocation strategies, parameter passing, dynamic storage allocation, Symbol table	(140 to 165)
Error Detecton & Recovery, Ad-Hoc and Systematic Methods	(166 to 168)
UNIT-IV : CODE GENERATION	
Intermediate code generation – Declarations, Assignment statements, Boolean expressions, Case statements, Back patching, Procedure calls	(169 to 203)
Code Generation – Issues in the design of code generator, Basic blocks and flow graphs, Register allocation and assignment	(203 to 216)
DAG representation of basic blocks, peephole optimization, generating code from DAG	(217 to 230)
UNIT-V : CODE OPTIMIZATION	
Introduction to Code optimization – Sources of optimization of basic blocks, loops in flow graphs, dead code elimination, loop optimization	(231 to 243)
Introduction to global data flow analysis, Code improving transformations, Data flow analysis of structure flow graph, Symbolic debugging of optimized code	(243 to 256)

UNIT

1

INTRODUCTION TO COMPILING AND LEXICAL ANALYSIS

INTRODUCTION OF COMPILER, MAJOR DATA STRUCTURE IN COMPILER, TYPES OF COMPILER, FRONT-END AND BACK-END OF COMPILER

Q.1. Explain the working of a compiler drawing its block diagram.
(R.G.P.V., June 2010)

Ans. A compiler is a system program that translates code written in high-level language (source language) like C or C++ into an equivalent low-level language (target language) like machine language as shown in fig. 1.1.

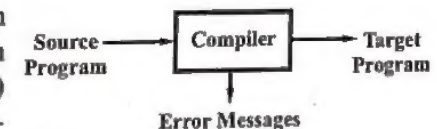


Fig. 1.1 A Compiler

At first glance, the variety of compilers may appear overwhelming. There are thousands of source languages. *Target languages* are equally as varied; a target language may be another programming language, or the machine language of any computer.

The translation from higher language to low level language is the primary job of the compiler. However, there are other important secondary functions that the compilers provide for helping the programmers develop software. Compilers are provided for reporting errors and warnings in the input higher language source to help the programmer in correcting them. Compiler allow options to help debug the execution of the executable program generated by it. Compilers offer options to generate extra 'profiling code' to report the statistics on the time taken by specific functions in the input source at run time. Today's compilers offer many other programmer-friendly features that help us develop software quickly and correctly, meeting all the specified requirements.

Compiler are sometimes classified as single-pass, multi-pass, load-and-go debugging or optimizing, depending on how they have been constructed or on what function they are supposed to perform.

Q.2. Describe the common data structures used by a compiler.

(R.G.P.V., June 2011)

Ans. The data structures that are needed by the phases as part of their operation and that serve to communicate information among the phases are as follows –

(i) **Tokens** – When a scanner collects characters into a token, it generally represents the token symbolically, that is, as a value of an enumerated data type representing the set of tokens of the source language. Sometimes it is also necessary to preserve the string of characters itself or other information derived from it, such as the name associated with an identifier token or the value of a number token.

(ii) **Syntax Tree** – If the parser generates a syntax tree, it is usually constructed as a standard pointer-based structure that is dynamically allocated as parsing proceeds. The entire tree can then be kept as a single variable pointing to the root node. Each node in the structure is a record whose fields represent the information collected both by the parser and, later, by the semantic analyzer. Sometimes, to save space, these fields are also dynamically allocated, or they are stored in other data structures, such as the symbol table, that allow selective allocation and deallocation.

(iii) **Symbol Table** – This data structure keeps information associated with identifiers – functions, variables, constants, and data types. The symbol table interacts with almost every phase of the compiler – the scanner, parser, or semantic analyzer may enter identifiers into the table; the semantic analyzer will add data type and other information; and the optimization and code generation phases will use the information provided by the symbol table to make appropriate object code choices. Since the symbol table will be accessed so frequently, insertion, deletion, and access operations need to be efficient, preferably constant-time operations. A standard data structure for this purpose is the hash table, although various tree structures can also be used. Sometimes several tables are used and maintained in a list or stack.

(iv) **Literal Table** – Quick insertion and lookup are essential as well to the literal table, which stores constants and strings used in a program. However, a literal table need not allow deletions, since its data applies globally to the program and a constant or string will appear only once in this table. The literal table is important in reducing the size of a program in memory by allowing the reuse of constants and strings. It is also needed by the code generator to construct symbolic addresses for literals and for entering data definitions in the target code file.

(v) **Intermediate Code** – Depending on the kind of intermediate code and the kinds of optimizations performed, this code may be kept as an array of text strings, a temporary text file, or as a linked list of structures. In compilers

that perform complex optimizations, particular attention must be given to choosing representations that permit easy reorganization.

(vi) **Temporary Files** – Historically, computers did not possess enough memory for an entire program to be kept in memory during compilation. This problem was solved by using temporary files to hold the products of intermediate steps during translation or by compiling “on the fly”, that is keeping only enough information from earlier parts of the source program to enable translation to proceed. Memory constraints are now a much smaller problem, and it is possible to require that an entire compilation unit be maintained in memory, especially if separate compilation is available in the language. Still, compilers occasionally find it useful to generate intermediate files during some of the processing steps. Typical among these is the need to *backpatch* addresses during code generation.

Q.3. Write short note on symbol table.

(R.G.P.V., June 2011)

Ans. Refer to Q.2 (iii).

Q.4. Write down the various types of compiler.

Ans. The types of compiler are as follows –

(i) **Incremental Compiler** – The compiler which compiles only the changed lines from the source code and update the object code.

(ii) **Cross Compiler** – The compiler used to compile a source code for different kinds platform.

(iii) **One Pass Compiler** – It is a type of compiler that compiles the whole process in only one-pass.

(iv) **Native Code Compiler** – The compiler used to compile a source code for same type of platform only.

(v) **Source to Source Compiler** – The compiler that takes high level language code as input and output source code of another high level language only.

(vi) **Threaded Code Compiler** – The compiler which simply replace a string by an appropriate binary code.

(vii) **Source Compiler** – The compiler which converts the source code high level language code into assembly language only.

Q.5. What is a ‘pass’ in a compiler? Differentiate between multiple pass compiler and a single pass compiler.

(R.G.P.V., June 2011)

Ans. In an implementation of a compiler, portions of one or more phases are grouped into a module known as *pass*. A pass reads the source program or the output of the previous pass, makes the transformations according to its phases and writes output into an intermediate file, which may then be read

through the next pass. Practically, there is great change in the way the phases of a compiler are grouped into passes.

Single Pass Compiler – These compilers defy all the considerations for a break-up into passes. Consequently, some restrictions have to be placed on the source language (e.g., declarations must precede use), and the memory requirements of the compiler are large. The code generated is also relatively inefficient. However, since, all the analysis and synthesis functions go hand in hand, time spent in constructing the intermediate code and scanning it in the subsequent pass(es) can be saved. These compilers have been found to be ten to fourteen times faster than their conventional counterparts, which makes them ideal for educational environments where program learning activity constitutes a considerable amount of the workload. Since such programs characteristically have short execution times, and running programs are rarely (if ever) re-submitted for execution, the relative inefficiency of the code is of minor importance.

Most of these systems are called ‘in-core batching compilers’ because they remain in core during the execution of the compiled program as well. This avoids the time spent in loading the compiler into memory for every job. These systems, therefore, also incorporate a batch-monitor for switching from one program of the batch to the next program.

Multipass Compiler – Multiple passes are required by a compiler, because when several phases are grouped into one pass, then the operation of the phases may be interleaved, with control alternating among several phases.

A multipass compiler can be made to use less space than a single pass compiler, since, the space occupied by the compiler program for one pass can be reused by the following pass. A multipass compiler is, of course, slower than a single pass compiler, because each pass reads and writes an intermediate file.

Q.6. Compare and contrast the features of a single pass compiler with multipass compiler. (R.G.P.V., Dec. 2010)

Ans. Refer to Q.5.

Q.7. Write the advantage of multipass compiler over single pass compiler. (R.G.P.V., Dec. 2015)

Ans. Refer to Q.5.

Q.8. Discuss the advantages and disadvantages of single pass and multipass compilers. (R.G.P.V., Dec. 2011, June 2012)

Ans. Refer to Q.5.

Q.9. Define pass of compiler. What are the factors that decide number of passes for a compiler? (R.G.P.V., June 2016)

Ans. Refer to Q.5.

The number of passes and the grouping of phases into passes are decided by a variety of reasons related to any particular programming language and machine, i.e. the structure of the source program is one of the deciding factor on the number of passes.

Certain languages require at least two passes to generate code easily. For example, languages such as PL/I or ALGOL68 allows the declaration of a name to occur after uses of that name. Code for expressions containing such a name cannot be generated conveniently until the declaration has been seen.

The environment in which the compiler must operate can also affect the number of passes. The compiler running on computers with small memory would normally use several passes while, on a computer with a large random access memory, a compiler with fewer passes would be possible.

Q.10. Write short note on bootstrapping. (R.G.P.V., June 2004, 2005, 2006, 2007, Dec. 2007, June 2008, Dec. 2008, 2011)

Or

Explain the process of bootstrapping used in compiler.

(R.G.P.V., June 2011)

Or

What do you mean by bootstrapping of compiler? (R.G.P.V., June 2017)

Ans. A compiler is complex enough program that we would like to write it in a friendlier language than assembly language. In the UNIX programming environment, compilers are usually written in C. Even C compilers are written in C. Using the facilities offered by a language to compile itself is the essence of *bootstrapping*. Bootstrapping is used to create compilers and to move them from one machine to another by modifying the back end.

For bootstrapping purposes, a compiler is characterized by three languages – the source language S that it compiles, the target language T that it generates code for, and the implementation language I that it is written in. We represent the three languages using the following fig. 1.2, called a *T-diagram*, because of its shape.

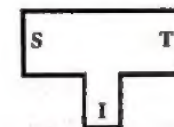


Fig. 1.2 T-diagram

T-diagram is abbreviated as $S_I T$. The three languages S, I and T may all be quite different. A compiler may run on one machine and produce target code for another machine such a compiler is called a cross compiler.

Suppose, we write a cross compiler for a new language L in implementation language S to generate code for machine N; that is, we create $L_S N$. If an existing compiler for S runs on machine M and generates code for M, it is characterized by $S_M M$. If $L_S N$ is

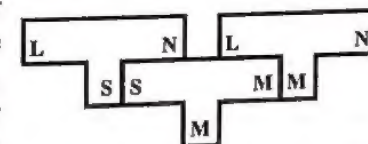


Fig. 1.3 Compiling a Compiler

run through $S_M M$, we get a compiler $L_M N$, that is, a compiler from L to N that runs on M. This process is illustrated in fig. 1.3 by putting together the T-diagrams for these compilers.

When T-diagrams are put together, the implementation language S of the compiler $L_S N$ must be the same as the source language of the existing compiler $S_M M$ and that the target language M of the existing compiler must be the same as the implementation language of the translated form $L_M N$. A trio of T-diagrams can be thought of as an equation –

$$L_S N + S_M M = L_M N$$

One form of bootstrapping builds up a compiler for larger and larger subsets of a language. Suppose, a new language L is to be implemented on machine M. Firstly, we write a small compiler that translates a subset S of L into the target code for M; that is, a compiler $S_M M$. We then use the subset S to write a compiler $L_S M$ for L. When $L_S M$ is run through $S_M M$, we obtain an implementation of L, $L_M M$.

For the advantages of bootstrapping to be realized fully, a compiler has to be written in the language it compiles. Suppose, we write a compiler $L_L N$ for language L in L to generate code for machine N. Development takes place on a machine M, where an existing compiler $L_M M$ for L runs and generates code for M. By first compiling $L_L N$ with $L_M M$, we obtain a cross compiler $L_M N$ that runs on M, but produces code for N. The compiler $L_L N$ can be compiled with the generated cross compiler. The result of second compilation is a compiler $L_N N$ that runs on N and generates code for N. The application of this two step process is shown in fig. 1.4.

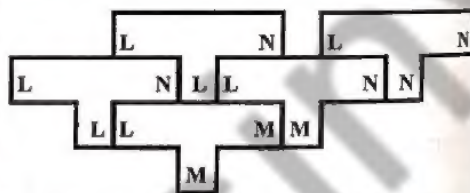


Fig. 1.4 Bootstrapping a Compiler

Q.11. What are major data structures used in compiler? Explain the concept of bootstrapping. (R.G.P.V., Nov. 2018)

Ans. Data Structures – Refer to Q.2.

Bootstrapping – Refer to Q.10.

Q.12. What is bootstrapping? Also explain cross compiler.

(R.G.P.V., Dec. 2015)

Ans. Bootstrapping – Refer to Q.10.

Cross Compiler – A cross compiler is a compiler that runs on one machine and produces object code for another machine, i.e., a $C_X^{(L \rightarrow Y)}$ Where $X \neq Y$. The cross compiler is used to implement the compiler, which is characterized

by three languages –

- (i) The source language
- (ii) The object language
- (iii) The language in which it is written.

The only difference between a cross compiler and a normal compiler is in terms of the code generated by it, no structural differences being warranted by this fact. Cross compilers are widely used for mini and micro computers because except for the architecture of machine Y, the factors influencing the design of $C_X^{(L \rightarrow Y)}$ are configuration of machine X, and not for machine Y. X is invariably a larger machine in these situations so that memory limitations on machine Y do not affect the compilation. This permits the use of HLLs for small machines which would otherwise be impossible.

Use of Cross Compiler for Transporting a Compiler of L to a New Machine – Let us assume that a cross compiler for machine B exists on machine A, i.e., $C_A^{(L \rightarrow B)}$ exists. Then we can obtain $C_B^{(L \rightarrow B)}$ if we develop a $C_L^{(L \rightarrow B)}$, i.e., a compiler for L written in L (see fig. 1.5).

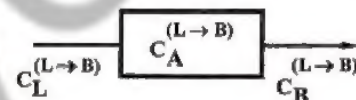


Fig. 1.5 Use of Cross Compiler to Transport a Compiler

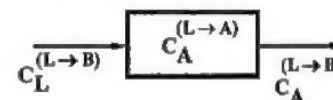


Fig. 1.6 The Bootstrapping Step in Transporting a Compiler

This looks like more work than coding $C_B^{(L \rightarrow B)}$ ourselves, since we have to code $C_L^{(L \rightarrow B)}$ and $C_A^{(L \rightarrow B)}$. However, this can be avoided by a bootstrapping step. What we do is to generate $C_A^{(L \rightarrow B)}$ as a result of bootstrapping with the use of $C_A^{(L \rightarrow A)}$. This can be obtained by compiling $C_L^{(L \rightarrow B)}$ under $C_A^{(L \rightarrow A)}$. The result is the compiler $C_A^{(L \rightarrow B)}$ we want (see fig. 1.6).

Now we review the steps of figs. 1.5 and 1.6 to determine what we need in order to obtain a compiler for L on machine B. We need a compiler of L written in L itself with B as the target machine, and we need a compiler for L on some other machine. Given the presence of such a compiler we need to code $C_L^{(L \rightarrow B)}$ in order to obtain $C_B^{(L \rightarrow B)}$, bootstrapping indeed. If the compiler for L is to be transported to many machines, we still need to do a lot of coding ($C_L^{(L \rightarrow X)}$ for all X). In order to further reduce the work involved, an intermediate language is often used in practice. Thus, instead of coding the various compilers for L (i.e., $C_X^{(L \rightarrow X)}$ for all X), a single compiler $C_L^{(L \rightarrow I)}$ is coded, where I is the intermediate language. In order to transport L to many

machines, we now have to code $C_L^{(L \rightarrow X)}$ for all X and append them to $C_L^{(L \rightarrow D)}$ to obtain the $C_L^{(L \rightarrow X)}$ we need (see fig. 1.7).

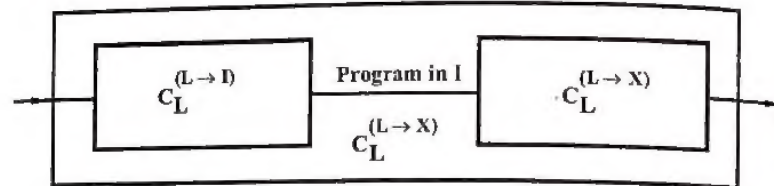


Fig. 1.7 Use of an Intermediate Language in Compiler Transportation

Q.13. What is a cross compiler ? Create a cross compiler C_S^{LA} using previously known language C_A^{SA} . (R.G.P.V., June 2012)

Ans. Cross Compiler – Refer to Q.12.

Cross Compiler for C_S^{LA} Using C_A^{SA} – To create a cross compiler C_S^{LA} by using C_A^{SA} , first, we assume that we write a compiler for a new language S in implementation language A to generate target code for machine A. Thus, the T diagram for this compiler is represented as shown in fig. 1.8.



Fig. 1.8

Now, if we run C_S^{LA} using C_A^{SA} then we get a compiler C_A^{LA} . That means a compiler for source language L that generates a target code in language A and which runs on machine A as shown in fig. 1.9.

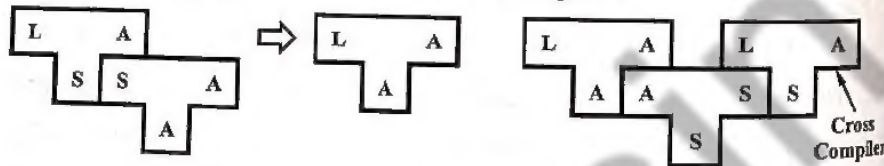


Fig. 1.9

By using this we obtain a cross compiler as shown in fig. 1.10.

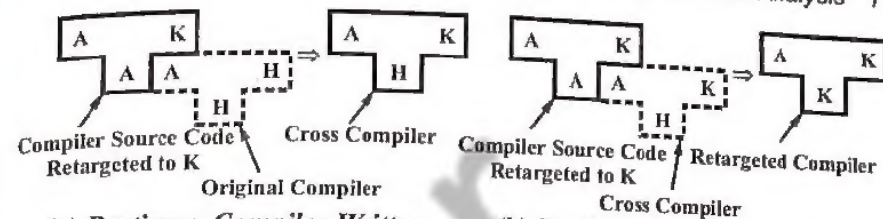
Q.14. Explain bootstrapping and porting. (R.G.P.V., Dec. 2013)

 O_T

Explain the concept of bootstrapping and porting in relation to compilation.
(R.G.P.V., Dec. 2016)

Ans. Bootstrapping – Refer to Q.10.

Porting – Porting the compiler to a new host computer now only requires that the back end of the source code be rewritten to generate code for the new machine. This is then compiled using the old compiler to produce a cross compiler, and the compiler is again recompiled by the cross compiler to produce a working version for the new machine. This is shown in fig. 1.11 (a) and (b).




(a) Porting a Compiler Written in its Own Source Language (Step 1) (b) Porting a Compiler Written in its Own Source Language (Step 2)

Fig. 1.11

Q.15. What is a translator? Compare compiler, assembler and interpreter.
(R.G.P.V., June 2012)

Ans. Translator – A translator is a program which performs translation into the machine code from the high level language of a computer. The translator plays a very important role in terms of the 'diagnostics' i.e., error-detection capability, supported by the translator. Thus, any violations of the high level language specification would be detected and informed to the programmer. As error reporting is in terms of the high level language program, it considerably simplifies the programmer's job. Fig. 1.12 shows the translation function of a translator.



```
graph LR; A[Source Program] --> B[Translator]; B --> C[Machine Language Program]
```

Fig. 1.12



Fig. 1.12

Comparison of Compiler, Assembler and Interpreter –

S. No.	Compiler	Interpreter	Assembler
(i)	A compiler translates a high-level language program into its equivalent machine language program.	A interpreter translates a high-level language program into its equivalent machine language program.	An assembler translates an assembly language program into its equivalent machine language program.
(ii)	The compiler translates each high-level language instruction into a set of machine language instructions, rather than a single machine language instruction. Hence there is a one-to-many correspondence between	It takes one statement of a high-level language program, translates it into machine language instructions and immediately executes it.	An assembler translates each assembly language instruction into an equivalent machine language instruction. Hence there is a one-to-one correspondence between the assembly language instructions of a source program and the

	the high-level language instructions of a source program, and the machine language instructions of its equivalent object program.	machine language instructions of a machine language instructions of its equivalent object program.
(iii)	It is a system software.	It is a system software.
(iv)	During the translation of a source program into its equivalent object program by the compiler, the source program is not being executed but it is only being converted into a form, which can be executed by the computer's processor.	During the translation of a source program into its equivalent object program by the assembler, the source program is not being executed but it is only being converted into a form, which can be executed by the computer's processor.

Q.16. Write the difference between compiler and interpreter.

(R.G.P.V., June 2017)

Ans. Refer to Q.15.

Q.17. What is front end and back end of compiler? What are the advantages of breaking up the compiler functionality into these two distinct stages?

(R.G.P.V., May 2018, 2019)

Ans. Front End – The front end analyzes the source code to build an internal representation of the program, called the intermediate representation. It also manages the symbol table, a data structure mapping each symbol in the source code to associated information such as location, type and scope.

While the front end can be a single monolithic function or program, as in a scannerless parser, it is more commonly implemented and analyzed as several phases, which may execute sequentially or concurrently. This method is favoured due to its modularity and separation of concerns. Most commonly today, the front end is broken into three phases – lexical analysis, syntax analysis and semantic analysis. Lexing and parsing comprise the syntactic analysis, and in simple cases these modules can be automatically generated from a grammar for the language, though in more complex cases these require manual modification. The lexical grammar and phrase grammar are usually context-free grammars, which simplifies analysis significantly, with context sensitivity handled at the semantic analysis phase. The semantic analysis phase is generally more complex and written by hand, but can be partially or fully automated using attribute grammars.

Back End – The back end is responsible for the CPU architecture specific optimizations and for code generation.

It is in routine to take the front end of a compiler and redo its associated back end to produce a compiler for the same source language on a different machine. If the back end of a compiler is designed carefully, it may not need to redesign too much back end. It is also possible to compile several different languages into the same intermediate language and use a common back end for different front ends, and hence obtaining several compilers for one machine.

COMPILER STRUCTURE – ANALYSIS-SYNTHESIS MODEL OF COMPILATION, VARIOUS PHASES OF A COMPILER

Q.18. Describe the analysis-synthesis model of compilation.

(R.G.P.V., Dec. 2013)

Ans. There are two parts of compilation – analysis and synthesis. The analysis part breaks up the source program into constituent pieces and creates an intermediate representation of the source program. The synthesis part constructs the desired target program from the intermediate representation. In these two parts, synthesis requires the most specialized techniques.

During analysis, the operations implied by the source program are determined and recorded in a hierarchical structure called a tree. Often, a special kind of tree called a syntax tree is used, in which each node represents an operation and the children of a node represent the arguments of the operation. For example, a syntax tree for an assignment statement is shown in fig. 1.13.

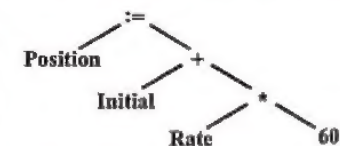


Fig. 1.13 Syntax Tree for
Position := Initial + Rate * 60

Many software tools that manipulate source programs first perform some kind of analysis. Some examples of these tools include –

(i) **Structure Editors** – A structure editor takes a sequence of commands as input to build a source program. The structure editor not only performs the text-creation and modification functions of an ordinary text editor, but it also analyzes the program text, putting an appropriate hierarchical structure on the source program. Thus, the structure editor can perform additional tasks that are useful in the preparation of programs. For example, it can check that the input is correctly formed, can supply keywords automatically, and can jump from a *begin* or left parenthesis to its matching *end* or right parenthesis. Further, the output of such an editor is often similar to the output of the analysis phase of compiler.

(ii) **Pretty Printers** – A pretty printer analyzes a program and prints it in such a way that the structure of the program becomes clearly visible. For example, comments may appear in a special font, and statements may appear with an amount of indentation proportional to the depth of their nesting in the hierarchical organization of the statements.

(iii) **Static Checkers** – A static checker reads a program, analyzes it, and attempts to discover potential bugs without running the program. The analysis portion is often similar to that found in optimizing compilers. For example, a static checker may detect that parts of the source program can never be executed, or that a certain variable might be used before being defined. In addition, it can catch logical errors such as trying to use a real variable as a pointer, employing the type-checking techniques.

(iv) **Interpreters** – Instead of producing a target program as a translation, an interpreter performs the operations implied by the source program. For an assignment statement, for example an interpreter might build a tree like fig. 1.13 and then carry out the operations at the nodes as it “walks” the tree. At the root it would discover it had an assignment to perform, so it would call a routine to evaluate the expression on the right, and then store the resulting value in the location associated with the identifier *position*.

Interpreters are frequently used to execute command languages, since each operator executed in a command language is usually an invocation of a complex routine such as an editor or compiler.

Traditionally, we think of a compiler as a program that translates a source language like Fortran into the assembly or machine language of some computer. The analysis portion in each of the following examples is similar to that of a conventional compiler.

(i) **Text Formatters** – A text formatter takes input that is a stream of characters, most of which is text to be typeset, but some of which includes commands to indicate paragraphs, figures, or mathematical structures like subscripts and superscripts.

(ii) **Silicon Compilers** – A silicon compiler has a source language that is similar or identical to a conventional programming language. However, the variables of the language represent, not locations in memory, but, logical signals (0 or 1) or groups of signals in a switching circuit. The output is a circuit design in an appropriate language.

(iii) **Query Interpreters** – A query interpreter translates a predicate containing relational and boolean operators into commands to search a database for records satisfying that predicate.

Q.19. Explain the following terms –

(i) **Translators, compiler and interpreters** (ii) **Bootstrapping.**
(R.G.P.V., Nov. 2018)

Ans. (i) Translators, Compiler and Interpreters – Refer to Q.15, Q.1 and Q.10.

(ii) **Bootstrapping** – Refer to Q.10.

Q.20. Explain various phases of a compiler.
(R.G.P.V., June 2008, Dec. 2012)

Or

Describe the different phases of compiler. (R.G.P.V., June 2005, 2006)

Or

Draw and explain various phases of a compiler.
(R.G.P.V., Dec. 2003, June 2007)

Or

Discuss the different phases in a compiler ? Explain each one of them.
(R.G.P.V., June 2011)

Or

Explain the various phases of compiler ? How phases of compilation converts the statement

$Position = initial + rate * 60$ (R.G.P.V., Dec. 2014)

Or

Discuss the various phases of compiler with the help of neat labelled diagram. (R.G.P.V., Dec. 2015)

Or

Explain various phases of compiler with the help of diagram.
(R.G.P.V., June 2016)

Or

What are the different phases of compiler ? Explain them with help of suitable example. (R.G.P.V., Dec. 2016)

Or

Explain the various phases of compiler with the help of diagram. Take any one example to elaborate complete working of compiler.
(R.G.P.V., Dec. 2017)

Or

What are the phases of a compiler ? Explain the function of each phase in brief with example. (R.G.P.V., Nov. 2018)

Ans. Phases of a Compiler – The compilation process includes several phases each of which transforms the source program from one representation to another. The block diagram, indicating all of these phases, is drawn in fig. 1.14.

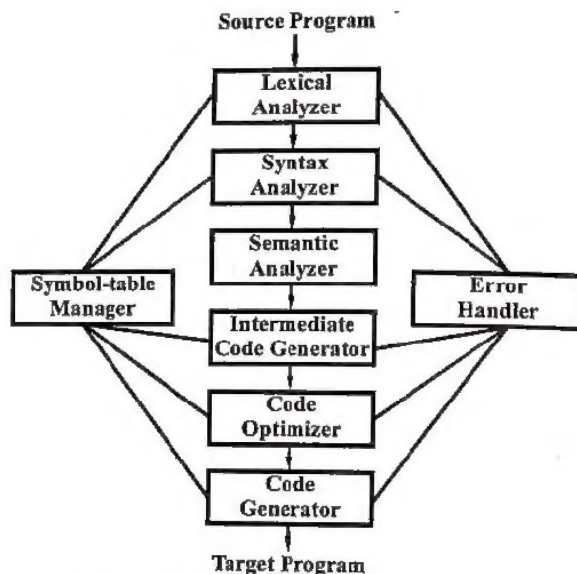


Fig. 1.14 Phases of a Compiler

The first three phases constitutes the analysis process and rest three makes the synthesis process. Two other activities, symbol table management and error handling, are shown interacting with six phases.

(i) **Symbol Table Management** – A symbol table is a data structure containing a record for each identifier, with fields for the attributes of the identifier. The data structure allows us to find the record for each identifier quickly and to store or retrieve data from that record quickly.

When an identifier in the source program is detected by the lexical analyzer, the identifier is entered into the symbol table. However, the attributes of an identifier cannot normally be determined during lexical analysis. The remaining phases enter information about identifiers into the symbol table and then use this information in various ways. The code generator enters and uses detailed information about the storage assigned to identifiers.

(ii) **Error Detection and Reporting** – Errors can encounter by each phase. The error handler not only correct simple errors but also generates an output to make the user aware of his mistakes and to take necessary actions, so that compilation can proceed.

The syntax and semantic analysis phases usually handle a large fraction of the errors detectable by the compiler. The lexical phase can detect errors where the characters remaining in the input do not form any token of the language. Errors where the token stream violates the structure rules (syntax) of the language are determined by the syntax analysis phase. During semantic

analysis the compiler tries to detect constructs that have the right syntactic structure but no meaning to the operation involved.

(iii) **Lexical Analysis** – In a compiler, linear analysis is called *lexical analysis or scanning*, in which the stream of characters making up the source program is read from left-to-right and grouped into tokens that are sequences of characters having a collective meaning. For example, in lexical analysis, the characters in the assignment statement

Position : = initial + rate * 60

would be grouped into the following tokens –

- | | |
|-----------------------------|-------------------------------|
| (a) The identifier position | (b) The assignment symbol : = |
| (c) The identifier initial | (d) The plus sign |
| (e) The identifier rate | (f) The multiplication sign |
| (g) The number 60. | |

The blanks separating the characters of these tokens would normally be eliminated during lexical analysis.

(iv) **Syntax Analysis** – Hierarchical analysis is called parsing or syntax analysis, in which characters or tokens are grouped hierarchically into nested collections with collective meaning.

Syntax analysis involves grouping the tokens of the source program into grammatical phrases that are used by the compiler to synthesize output. Usually, the grammatical phrases of the source program are represented by a parse tree.

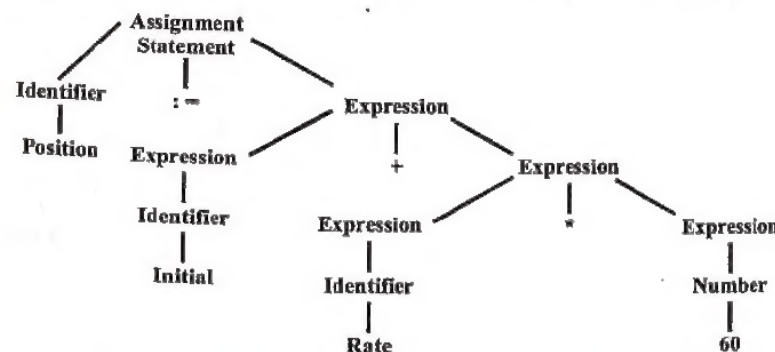


Fig. 1.15 Parse Tree for Position : = Initial + Rate * 60

(v) **Semantic Analysis** – There are certain checks to be performed to ensure that the components of a program fit together meaningfully. The semantic analysis phase checks the source program for semantic errors and gathers type information for the subsequent code generation phase. It uses the hierarchical structure determined by syntax analysis phase to identify the operators and operands of expressions and statements.

An important component of semantic analysis is type-checking. The compiler checks that each operator has operands that are permitted by source language specification. For example, many programming language definitions require a compiler to report an error every time a real number is used to index an array. However the language specification may permit some operand coercions, for example, when a binary arithmetic operator is applied to an integer and real. In this case, compiler may need to convert the integer to a real.

(vi) Intermediate Code Generation – After syntax and semantic analysis, some compilers generate an explicit intermediate representation of the source program. The intermediate representation should have two important properties – it should be easy to produce, and easy to translate into the target program. An intermediate form called “Three-address code” which is like the assembly language for a machine in which every memory location can act like a register. Three-address code consists of a sequence of instructions, each of which has at most three operands. The intermediate form has several properties. First, each three-address statement has at most one operator in addition to the assignment. Second, the compiler must generate a temporary name to hold the value computed by each instruction. Third, some “three-address” instructions have fewer than three operands, e.g., the first and last instructions of instructions shown below –

```
temp 1 := inttoreal (60)
temp 2 := id3 * temp 1
temp 3 := id2 + temp 2
id1 := temp 3
```

(vii) Code Optimization – The code optimization phase attempts to improve the intermediate code, so that faster-running machine code will result. The codes are optimized so as to simply make the execution process short. For example –

```
temp 1 := id3 * 60.0
id1 := id2 + temp 1
```

There is nothing wrong with this simple algorithm, since the problem can be fixed during the code optimization phase. That is, the compiler can deduce that the conversion of 60 from integer to real representation can be done once and for all at compile time, so the inttoreal operation can be eliminated.

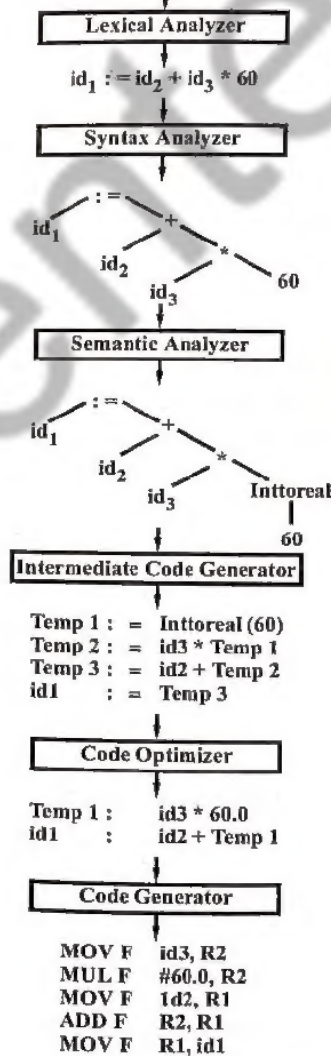
There are simple optimizations that significantly improve the running time of the target program without slowing down compilation too much.

(viii) Code Generation – The final phase of compiler is the generation of target code, consisting normally of relocatable machine code or assembly code. Memory locations are selected for each of the variables used by the program. Then, intermediate instructions are each translated into a sequence of machine instructions that perform the same task.

For example –

```
MOV F id3, R2
MUL F # 60.0, R2
MOV F id2, R1
ADD F R2, R1
MOV F R1, id1
```

Position := Initial + Rate * 60



Symbol Table		
1	Position	...
2	Initial	...
3	Rate	...
4		

Fig. 1.16

The first and second operands of each instruction specify a source and destination. The F in each instruction tells us that instructions deal with floating-

point numbers. This code moves the contents of the address id3 into register 2, then multiplies it with the real constant 60.0. The # signifies that 60.0 is to be treated as a constant. The third instruction moves id2 into register 1 and adds to it the value previously computed in register 2. Finally, the value in register 1 is moved into the address of id1, so the code implements the assignment as shown in fig. 1.16.

Q.21. Discuss the importance of symbol table in compiler design. How is the symbol table manipulated at various phases of compilation?

(R.G.P.V., Dec. 2012)

Ans. Refer to Q.2 (iii) and Q.20.

Q.22. What are the tasks performed by the compiler in the lexical and syntax analysis phases?

(R.G.P.V., Dec. 2007, 2010)

Or

Discuss the various tasks performed by the compiler in the lexical and syntax analysis phase.

(R.G.P.V., Dec. 2011)

Ans. Refer to Q.20 (iii) and (iv).

Q.23. How syntax analyzer generates token? (R.G.P.V., June 2016)

Ans. Syntax analysis is the second phase of compiler syntax analysis and also known as parsing. Parsing is the process of determining whether a string of token can be generated by a grammar. It is performed by syntax analyzer which can also be termed as parser. In addition to construction of the parse tree, syntax analysis also checks and reports syntax errors accurately. Parser is a program that obtains tokens from lexical analyzer and constructs the parse tree which is passed to the next phase of compiler for further processing. Parser implements context free grammar for performing error checks.

Q.24. Briefly explain the compiler construction tools.

(R.G.P.V., Dec. 2014)

Or

Write short note on compiler writing tools. (R.G.P.V., Dec. 2011)

Ans. The tools that are needed in compiler construction are as follows –

(i) **Parser Generator** – These produce syntax analyzers normally from input that is based on a context free grammar. In early compilers syntax analysis consumed not only a large fraction of the running time of a compiler, but a large fraction of the intellectual effort of writing a compiler. This phase is now considered one of the easiest to implement.

(ii) **Scanner Generator** – These automatically generate lexical analyzers, normally from a specification based on regular expressions. The basic organization of the resulting lexical analyzer is in effect a finite automaton.

(iii) **Syntax Directed Translation Engines** – These produce collections of routines that walk the parse tree. The basic idea is that one or more “translations” are associated with each node of the parse tree, and each translation is defined in terms of translations at its neighbour nodes in the tree.

(iv) **Automatic Code Generators** – Such a tool takes a collection of rules that defines the translation of each operation of the intermediate language into the machine language for the target machine. The rules must include sufficient detail that we can handle the different possible access methods for data, e.g., variables may be in register, in a fixed location in memory, or may be allocated a position on a stack. The basic technique is “*template matching*”. The intermediate code statements are replaced by “templates” that represent sequence of machine instructions, in such a way that the assumptions about storage of variables match from template to template. Since, there are usually many options regarding where variables are to be placed, there are many possible ways to “tile” intermediate code with a given set of templates, and it is necessary to select a good tiling without a combinatorial explosion in running time of the compiler.

(v) **Data Flow Engine** – Much of the information needed to perform good code optimization involves “data flow analysis”, the gathering of information about how values are transmitted from one part of a program to each other part. Different task of this nature can be performed by essentially, the same routine, with the user supplying details of the relationship between intermediate code statements and the information being gathered.

LEXICAL ANALYSIS – INPUT BUFFERING, SPECIFICATION & RECOGNITION OF TOKENS, DESIGN OF A LEXICAL ANALYZER GENERATOR, LEX

Q.25. Explain lexical analyzer. What is the role of the lexical analyzer in compiler?

Ans. The lexical analyzer is the first phase of a compiler. Its main task is to read the input characters and produce as output, a sequence of tokens that the parser uses for syntax analysis. As shown in fig. 1.17 upon receiving a “get next token” command from the parser, the lexical analyzer reads input characters until it can identify the next token.

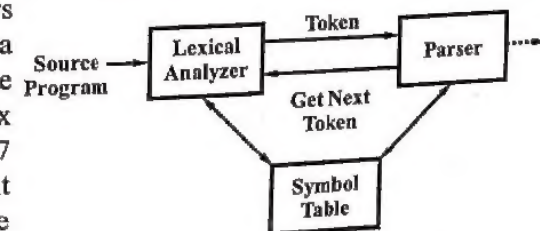


Fig. 1.17 Interaction of Lexical Analyzer with Parser

Since, the lexical analyzer is the part of the compiler that reads the source text, it may also perform certain secondary tasks at the user interface. Some tasks that are performed by lexical analyzer are stripping out from source program comments and white space in the form of blank, tab and newline characters. Another is correlating error messages from the compiler with the source program. In some compilers, the lexical analyzer is in charge making a copy of the source program with the error messages marked in it. If source language supports some macro preprocessor functions, then these preprocessor functions may also be implemented during lexical analysis.

Sometimes, lexical analyzers are divided into a cascade of two phases, the first called "*scanning*", and the second "*lexical analysis*". The scanner is responsible for doing simple tasks, while the lexical analyzer performs the more complex operations.

The interface of a lexical analyzer can be shown in fig.1.18. It reads characters from the input, groups them into lexemes and passes the tokens formed by the lexemes, together with their attributes values, to the later stages of compilers. In some cases, lexical analyzer has to read some characters ahead before it can decide on the token to be returned to the parser. The extra character has to be pushed back onto the input, because, it can be the beginning of next lexeme in the input.

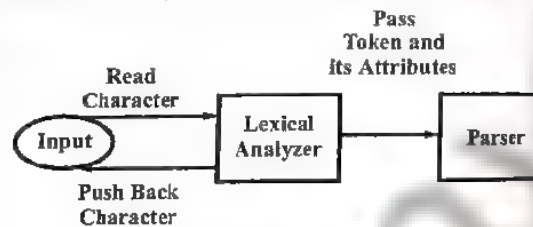


Fig. 1.18 Inserting a Lexical Analyzer between the Input and the Parser

Lexical analyzer build some useful meaning word and considers them as **TOKEN** which are defined as sequence of characters having a collective meaning. These tokens are then passed by analyzer to the next phase called **syntax analyzer**. There are varieties of tokens each have certain information related bits called **ATTRIBUTES**.

Depending upon the types of token each type thus have a specific pattern. For a given grammar token types are fixed for a valid program. Output of a lexical analyzer called **lexeme**. The lexical analyzer produces tokens and the parser consumes them and form a producer-consumer pair.

Q.26. Explain the functions of lexical analyzer in brief.

Ans. Refer to Q.25.

Q.27. Draw the block diagram of lexical and syntax analyzer and state clearly input, output and task performed by them.

Ans. Lexical Analyzer – Refer to Q.25.

Syntax Analyzer – The parser obtains a string of tokens from the lexical analyzer and verifies that the string can be generated by the grammar for the source language. A parser also report any syntax errors in an intelligible fashion. It should also recover from commonly occurring errors so that it can continue processing the remainder of its input.

This is the function of the syntax analyser. It uses the concept of derivation and reduction applied on the context free grammar. There are number of tasks that are performed during parsing such as collecting information about various tokens into the symbol table, performing type checking and semantic analysis and intermediate code.

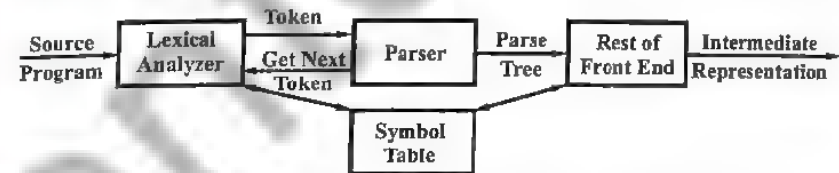


Fig. 1.19 Position of Parser in Compiler Model

Q.28. Define finite automata.

Ans. A finite automata consists of a finite number of states and a finite number of transitions, and these transitions are defined on certain, specific symbols called input symbols. One of the states of the finite automata is identified as the initial state, the state in which the automata always starts, similarly, certain states are identified as final states.

Q.29. What do you understand by automatic lexical generator ?

(R.G.P.V., Dec. 2012)

Ans. An automatic lexical generator is a tool that is used to generate a code to perform lexical analysis of the input, given the rules for the basic building blocks of the language. These rules of the language are known as its lexical specifications. The lexical specifications of the language are passed to the automatic lexical generator as shown in fig. 1.20. The automatic lexical generator then transforms the lexical specifications into a lexical analyzer that can be used to tokenize an input.

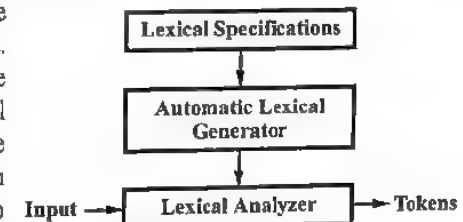


Fig. 1.20

An automatic lexical generator is used to make easy the development of lexical analyzers for any language. This type of generator helps to over-come the following problems –

(i) There are various design and coding effort goes into parsing of the input that could be same for lexical analyzers of any programming language.

(ii) To create a lexical analyzer for a new language would be difficult and complicated.

(iii) The complexity of the lexical analyzer would be very high and adding a new construct to an existing language could become difficult.

Q.30. Give the reasons for the separation of scanner and parser in separate phases of compiler. (R.G.P.V., Dec. 2005)

Or

What are various issues in lexical analyzer ?

Ans. There are several reasons for separating the analysis phase of compiling into scanners (lexical analysis) and parsing (syntax analysis).

(i) **Simpler Design** – Simpler design is the most important consideration. The separation of lexical analysis from syntax analysis allows us to simplify one or other of these phases. For example a parser embodying the conventions for comments and white space is more complex than one that can assume comments and white space have already been removed by a lexical analyzer.

(ii) **Clean Overall Language Design** – If we are designing a new language, separating the lexical (scanner) and parsing conventions can lead to a cleaner overall language design.

(iii) **Compiler Efficiency** – A separate lexical analyzer allows us to construct a specialized and potentially more efficient processor for the task. Specialized buffering techniques for reading input characters and processing tokens can significantly speed up the performance of a compiler. Thus the compiler efficiency is improved.

(iv) **Compiler Portability** – Portability is enhanced. Input alphabet peculiarities and other device-specific anomalies can be restricted to the lexical analyzer. The representation of special or non standard symbols can be isolated in lexical analyzer e.g., symbol ↑ in Pascal.

Q.31. Why the analysis portion of a compiler is normally separated into lexical analysis and parsing phase ? Explain tokens, patterns and lexemes with an example. (R.G.P.V., May 2019)

Ans. Refer to Q.30.

Token – Sequence of character having a collective meaning is known as token.

Typical tokens are,

(i) identifiers (ii) keywords (iii) operators (iv) special symbols (v) constants.

Lexeme – The character sequence forming a token is called lexeme for token.

Pattern – The set of rules by which set of string associate with single token is known as pattern. The examples of token, lexeme and pattern are shown in table 1.1.

Table 1.1

Token	Lexeme	Pattern
id	x y n 0	letter followed by letters and digits
number	3.14159, 0, 6.02e23	any numeric constant
If	If	If
relation	<, <=, =, >, >=	< or <= or = or > or >= or letter followed by letters & digit
Literal	"abcxyz"	anything but ", surrounded by" 's

For Example – (i) if (x <= 5) : Token – if (keyword),

X (id),

<= (relation),

5 (number)

Lexeme – if, X, <= 5

Q.32. Explain the role of the lexical analysis in compiler and also discuss the issues in lexical analysis. (R.G.P.V., Dec. 2013)

Ans. Refer to Q.25 and Q.30.

Q.33. Explain the concept of input buffering. (R.G.P.V., Dec. 2006, 2009)

Or

Explain input buffering. What is the need for buffer pairs ?

(R.G.P.V., Dec. 2004)

Or

Explain input buffering.

(R.G.P.V., Dec. 2013)

Or

What do you mean by buffering ?

(R.G.P.V., June 2016)

Or

Explain the term input buffering in brief.

(R.G.P.V., Dec. 2016)

Or

What is meant by input buffering ? Explain the use of sentinels in recognizing tokens. (R.G.P.V., Dec. 2017)

Ans. Input buffering is done for increasing efficiency of compiler. A two-buffer input scheme is useful when look-ahead on the input is necessary to identify tokens. To mark the end of the buffer we use *sentinels* and this technique is useful for speeding up the lexical analyzer. There are three general approaches to the implementation of a lexical analyzer –

(i) Use a lexical-analyzer generator, such as the Lex compiler to produce the lexical analyzer from a regular-expression based specification.

In this case, the generator provides routines for reading and buffering the input.

(ii) Write the lexical analyzer in a conventional systems-programming language, using the I/O facilities of that language to read the input.

(iii) Write the lexical analyzer in assembly language and explicitly manage the reading of input.

The three choices are listed in order of increasing difficulty for the implementor. Unfortunately, the harder-to-implement approaches often yield faster lexical analyzers. Since the lexical analyzer is the only phase of the compiler that reads the source program character-by-character, it is possible to spend a considerable amount of time in the lexical analysis phase, even though the later phases are conceptually more complex. Thus, the speed of lexical analysis is a concern in compiler design.

For many source languages, there are times when the lexical analyzer needs to look ahead several characters beyond the lexeme for a pattern before a match can be announced. The lexical analyzers used a function *ungetc* to push lookahead characters back into the input stream. Because a large amount of time can be consumed moving characters, specialized buffering techniques have been developed to reduce the amount of overhead required to process an input character. Fig. 1.21 shows a buffer divided into two N-character halves. Typically, N is the number of characters on one disk block, e.g., 1024 or 4096. We read N input characters into each half of the buffer with one system read command, rather than invoking a read command for each input character. If fewer than N characters remain in the input, then a special character *eof* is read into the buffer after the input characters. That is, *eof* marks the end of the source file and is different from any input character.

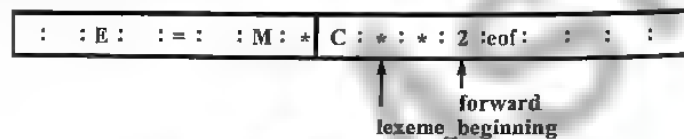


Fig. 1.21 An Input Buffer in Two Halves

Two pointers to the input buffer are maintained. The string of characters between the two pointers is the current lexeme. Initially, both pointers point to the first character of the next lexeme to be found. One, called the forward pointer, scans ahead until a match for a pattern is found. Once the next lexeme is determined, the forward pointer is set to the character at its right end. After the lexeme is processed, both pointers are set to the character immediately past the lexeme. With this scheme, comments and white space can be treated as patterns that yield no token. If the forward pointer is about to move past the halfway mark, the right half is filled with N new input characters. If the

forward pointer is about to move past the right end of the buffer, the left half is filled with N new characters and the forward pointer wraps around to the beginning of the buffer. This buffering scheme works quite well most of the time, but with it the amount of lookahead is limited, and this limited lookahead may make it impossible to recognize tokens in situations where the distance that the forward pointer must travel is more than the length of the buffer.

Q.34. Describe the role of a lexical analyzer and also explain the concept of input buffering.
(R.G.P.V., June 2009, Dec. 2010)

Ans. Refer to Q.25 and Q.33.

Q.35. Explain how tokens are specified.

Ans. For specifying patterns, regular expressions are an important notation. A set of strings is matched by every pattern, so for sets of strings, regular expressions will serve as names.

String and Language – Any finite set of symbols is denoted by the term alphabet or character class. Letters and characters are type typical examples of symbols. The set {0, 1} is the binary alphabet. Two examples of computer alphabets are ASCII and EBCDIC. In language theory, the terms sentence and word are generally used as synonyms for the term string. The length of string S is represented by |S|. Length is determined according to number of symbol in string S.

For Example – Apple is a string of length five “e” – denote empty string. Empty string, denoted by ϵ , is a special string of length zero. Some common terms associated with parts of a string are given below –

S.No.	Term	Definition
(i)	Subsequence of S	Any string formed by deleting zero or not necessarily contiguous symbols from S. Example – Appe is a subsequence of apple.
(ii)	Proper prefix, suffix, or substring of S.	Any non empty string Z that is, respectively, a prefix, suffix, or substring of S like that $S \neq z$
(iii)	Substring of S	A string obtained by deleting a suffix and a prefix from S.
(iv)	Suffix of S	A string formed by deleting zero or more of the leading symbols of S.
(v)	Prefix of S	A string obtained by removing zero or more trailing symbols of string S.

Operations on Languages – There are some important operations which can be applied to languages. For lexical analysis, we are interested firstly in union, concatenation, closure.

S.No.	Operation	Definition
(i)	Positive closure of L written L^+	$L^+ = \bigcup_{i=1}^{\infty} L^i$ L^+ denotes one or more concatenations of L
(ii)	Kleene closure of L written L^*	$L^* = \bigcup_{i=0}^{\infty} L^i$ L^* denotes zero or more concatenations of L
(iii)	Concatenation of L and M written LM	$LM = \{St/S \text{ is in } L \text{ and } t \text{ is in } M\}$
(iv)	Union of L and M written LUM	$L \cup M = \{S/S \text{ is in } L \text{ or } S \text{ is in } M\}$

Regular Expressions – The languages accepted by finite automata are easily described by simple expressions called regular expressions. Regular expressions are useful for representing certain sets of strings in an algebraic fashion. This notation of regular expression involves a combination of strings of symbols from some alphabet over Σ , parenthesis, and the operators such as $+$ and $*$.

Let Σ be a finite set of symbols and let L, L_1 and L_2 be sets of strings from Σ^* . The concatenation, of L_1 and L_2 , denoted by L_1L_2 , is the set $\{xy | x \text{ is in } L_1 \text{ and } y \text{ is in } L_2\}$ and the union of L_1 and L_2 , denoted by $L_1 + L_2$, is the set $\{x + y | x \text{ is in } L_1 \text{ and } y \text{ is in } L_2\}$. The Kleene closure (or just closure) of L, denoted by L^* , is the set

$$L^* = \bigcup_{i=0}^{\infty} L^i$$

and the positive closure of L, denoted by L^+ , is the set

$$L^+ = \bigcup_{i=1}^{\infty} L^i$$

That is, L^* denotes words constructed by concatenating any number of words from L. L^+ is the same, but the case of zero words, whose "concatenation" is defined to be ϵ , is excluded.

The regular expressions over Σ and the sets that they denote are defined recursively as follows –

(i) Any terminal symbol (i.e., an element of Σ), Λ , ϕ , are regular expressions. These are called **primitive regular expressions**

(ii) The union of two regular expressions R_1 and R_2 , written $R_1 + R_2$, is also a regular expression

(iii) The concatenation of two regular expressions R_1 and R_2 , written as R_1R_2 , is also a regular expression

(iv) The iteration (or closure) of a regular expression R, written as R^* , is also a regular expression

(v) If R is a regular expression, then (R) is also a regular expression. In the absence of parenthesis, we have the hierarchy of operations as follows – iteration (closure), concatenation, and union. i.e., in evaluating a regular expression involving various operations, we perform iteration first, then concatenation, and finally union.

Following are the identities for regular expressions. These are useful for simplifying regular expressions –

- (i) $\phi + R = R$
- (ii) $\phi R = R\phi = \phi$
- (iii) $\Lambda R = R\Lambda = R$
- (iv) $\Lambda^* = \Lambda$ and $\phi^* = \Lambda$
- (v) $R + R = R$
- (vi) $R^*R^* = R^*$
- (vii) $RR^* = R^*R$
- (viii) $(R^*)^* = R^*$
- (ix) $\Lambda + RR^* = R^* = \Lambda + R^*R$
- (x) $(PQ)^*P = P(QP)^*$
- (xi) $(P + Q)^* = (P^*Q^*)^* = (P^* + Q^*)^*$
- (xii) $(P + Q)R = PR + QR$ and $R(P + Q) = RP + RQ$.

Regular Definitions – For notational convenience, we can regard to give names to regular expressions and to define regular expressions using these names as if they were symbols.

A regular definition is a sequence of definitions if Σ is an alphabet of basic symbols

$$D_1 \rightarrow R_1$$

$$D_2 \rightarrow R_2$$

$$\dots\dots\dots$$

$$D_n \rightarrow R_n$$

Here, D_i represents distinct name and R_i represents regular expression.

Notational Shorthands – In regular expression, certain constructs occur so frequently. It is convenient to introduce notational shorthands for them.

(i) **One or more Instances** – The unary postfix operator "+" means one or more instances of. The operator "+" and "*" have same precedence and associativity.

(ii) **Zero or one instance** – The unary postfix operator “?” means zero or one instance of.

(iii) **Character Classes** – The notation $[abc]$, here a , b and c are alphabet symbols represents regular expression $a | b | c$. An abbreviated character class like $[a - z]$ represents regular expression $a|b|...|z$. By using character classes, we can define identifiers as being strings generated by the regular expression

$[A - Za - Z][A - Za - Z0 - 9]^*$

Non regular sets –

$\{WCW | W \text{ is a string of } a'S \text{ and } b'S\}$

The set can be denoted by neither regular expression nor context-free grammar

Q.36. Explain how tokens are recognized. (R.G.P.V., June 2005, 2006)

Ans. The recognition of token is done to separate out different tokens. To understand the concept let us consider the following grammar fragment –

$\text{stmt} \rightarrow \text{if expr then stmt} | \text{if expr then stmt else stmt} | \epsilon$

$\text{expr} \rightarrow \text{term relop term} | \text{term}$

$\text{term} \rightarrow \text{id} | \text{num}$

where the terminals **if**, **then**, **else**, **relop**, **id**, **num** letter and **digit** generate sets of strings given by the following regular definitions –

$\text{if} \rightarrow \text{if}$

$\text{then} \rightarrow \text{then}$

$\text{else} \rightarrow \text{else}$

$\text{relop} \rightarrow < | < = | = | > | > =$

$\text{id} \rightarrow \text{letter} (\text{letter} | \text{digit})^*$

$\text{num} \rightarrow \text{digit}^+ (\text{digit}^+ | \text{E}(\text{+|-}) | \text{digit}^+)$

For this language fragment the lexical analyzer will recognize the keywords **if**, **then**, **else** as well as the lexemes denoted by **relop**, **id**, and **num**. To simplify matters, we assume keywords are reserved; that is, they cannot be used as identifiers. In addition, we assume lexemes are separated by white space consisting of nonnull sequences of blanks, tabs, and newlines. Our lexical analyzer will strip out white space. It will do so by comparing a string against the regular definitions WS below –

$\text{delim} \rightarrow \text{blank/tab/newline}$

$\text{ws} \rightarrow \text{delim}^+$

If a match for **ws** is found, the lexical analyzer does not return a token to the parser. Rather, it proceeds to find a token following the white space and returns that to the parser. Our goal is to construct a lexical analyzer that will isolate the lexeme for the next token in the input buffer and produce as output a pair consisting of the appropriate token and attribute value, using the translation

table given in fig. 1.22. The attribute-values for the relational operators are given by the symbolic constants LT, LE, EQ, NE, GT, GE.

Regular Expression	Token	Attribute Value
ws	—	—
if	if	—
then	then	—
else	else	—
id	id	pointer to table entry
num	num	pointer to table entry
<	relop	LT
< =	relop	LE
=	relop	EQ
>	relop	NE
> =	relop	GT
	relop	GE

Fig. 1.22 Regular Expression Patterns for Token

Q.37. What are the issues in lexical analysis? Explain in detail the recognition of tokens. (R.G.P.V., Dec. 2014)

Ans. Refer to Q.30 and Q.36.

Q.38. What is simple approach to the design of lexical analyzer for an identifier? (R.G.P.V., Dec. 2007)

Or

Discuss in detail, the approach to the design of lexical analyzer with example. (R.G.P.V., Dec. 2009)

Ans. One way to begin the design of any program is to describe the behaviour of the program by a flowchart. This approach is particularly useful when the program is a lexical analyzer, because the action taken is highly dependent on what characters have been seen recently. This specialized kind of flowchart for lexical analyzers, called a *transition diagram*. In a transition diagram, the boxes of the flowchart are drawn as circles and called states. The states are connected by arrows, called edges. The labels on the various edges leaving a state indicate the input characters that can appear after the state.

Fig. 1.23 shows a transition diagram for an identifier, defined to be a letter followed by any number of letters or digits. The starting state of the transition diagram is state 0, the edge from which indicates that the first input character must be a letter. If this is the case, we enter state 1 and look at the next input character. If that is a letter or digit, we re-enter state 1 and look at

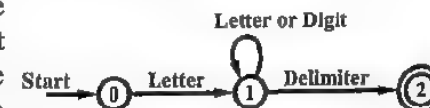


Fig. 1.23 Transition Diagram for

the input character after that. We continue this way, reading letters and digits and making transitions from state 1 to itself, until the next input character is a delimiter for an identifier, which here we assume is any character that is not a letter or a digit. On reading the delimiter, we enter state 2.

Consider again the transition diagram shown in fig. 1.23. The code for state 0 might be –

```
State 0 : C := GETCHAR();
          if LETTER(C) then goto state 1
          else FAIL( )
```

Here, LETTER is a procedure which returns **true** if and only if C is a letter. FAIL is a routine which retracts the lookahead pointer and starts up the next transition diagram, if there is one, or calls the error routine. The code for state 1 is –

```
State 1 : C := GETCHAR();
          if LETTER(C) or DIGIT(C) then goto state 1
          else if DELIMITER(C) then goto state 2
          else FAIL( )
```

DIGIT is a procedure which returns **true** if and only if C is one of the digits 0, 1, ..., 9. DELIMITER is a procedure which returns **true** whenever C is a character that could follow an identifier.

State 2 indicates that an identifier has been found. Since the delimiter is not part of the identifier, we must retract the lookahead pointer one character for which we use a procedure RETRACT. We use a * to indicate states of which input retraction must take place. In state 2 we return to the parser a pair consisting of the integer code for an identifier, which we denote by id and a value that is a pointer to the symbol table returned by INSTALL. The code for state 2 is –

```
State 2 : RETRACT();
          return (id, INSTALL( ))
```

Q.39. What is LEX? Describe auxiliary definitions and translation rule for LEX with suitable example. (R.G.P.V., Dec. 2005, June 2007, 2008, Dec. 2010, 2011, June 2012, Dec. 2013)

Ans. LEX – It is the tool that has been widely used to specify lexical analyzers for a variety of languages. This tool is referred to as **LEX compiler** and its input specification is called **LEX Language**. A LEX source program is a specification of a lexical analyzer, consisting of a set of regular expressions together with an action for each regular expression. The output of LEX is lexical analyzer program constructed from the LEX source specification.

LEX can be a compiler for a language that is very good for writing lexical analyzers and some other tasks of a text-processing nature. Most programming languages, a source program for LEX does not supply all the details of the intended computation. Rather, LEX itself supplies with its output a program that simulates a finite automaton. This program takes a transition table as data. The transition table is that portion of LEX's output that stems directly from LEX's input. The situation is shown in fig. 1.24, where the lexical analyzer L is the transition table plus the program to simulate an arbitrary finite automaton expressed as a transition table.

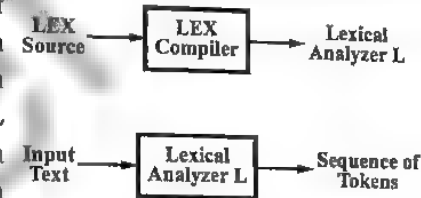


Fig. 1.24 The Role of LEX

LEX Specification – A Lex program consists of three parts –

```
declarations
%%
translation rules
%%
auxiliary procedures
```

The declaration includes declarations of variables, manifest constants, and regular definition. The auxiliary definitions are statements of the form

$$D_1 = R_1$$

$$D_2 = R_2$$

.

.

.

.

$$D_n = R_n$$

where each D_i is a distinct name, and each R_i is a regular expression whose symbols are chosen from $\Sigma \cup \{D_1, D_2, \dots, D_{i-1}\}$, i.e., characters of previously defined names. The D_i 's are shorthand names for regular expressions. Σ is our input symbol alphabet, e.g., the ASCII or EBCDIC character sets.

For example, we can define the class of identifiers for a typical programming language with the following sequence of auxiliary definitions.

letter = A | B | | Z

digit = 0 | 1 | | 9

identifier = letter (letter | digit)*

The translation rules of a LEX program are statements of the form

P_1 {action₁}
 P_2 {action₂}

 P_n {action_n}

where each P_i is a regular expression called a *pattern* and each action *action_i* is a program fragment describing what action the lexical analyzer should take when lexeme matched by pattern P_i is found in the input. The *action_i*'s are written in a conventional programming language. To create the lexical analyzer L, each of the *action_i*'s must be compiled into machine code, just like any other program written in the language of the *action_i*'s.

A lexical analyzer L created by Lex behaves in concert with a parser. When activated by the parser, the L begins reading its input, one character at a time until it has found the longest prefix of the input that is matched by one of the regular expressions P_i . Once L has found that prefix, L removes it from the input and places it in a buffer called TOKEN. L then executes the action *action_i*. After completing *action_i*, L returns control to the parser. When requested to, L repeats this series of actions on the remaining input. It is possible, however, that none of the regular expressions denoting the tokens matches any prefix of the input. In that case, an error has occurred, and L presumably transfers control to some error handling routine. It is also possible that two or more patterns match the same longest prefix of the remaining input. If that is the case, L will break the tie in favor of the token which came first in the list of translation rules. The repeated search for lexemes until an explicit return allows the lexical analyzer to process white space and comments conveniently. The lexical analyzer returns a single quantity, the token, to the parser. To pass an attribute value with information about the lexeme, we can set a global variable called yyval.

For example, consider the collection of tokens as shown in fig. 1.25. The translation rules for these tokens is defined in fig. 1.26.

Token	Code	Value
begin	1	—
end	2	—
if	3	—
then	4	—
else	5	—
identifier	6	Pointer to symbol table
constant	7	Pointer to symbol table
<	8	1
<=	8	2
=	8	3
<>	8	4
>	8	5
>=	8	6

Fig. 1.25 Tokens Recognized

AUXILIARY DEFINITIONS

letter = A | B | | Z
 digit = 0 | 1 | | 9

TRANSLATION RULES

BEGIN	{return 1}
END	{return 2}
IF	{return 3}
THEN	{return 4}
ELSE	{return 5}
letter(letter digit)*	{LEXVAL : = INSTALL(); return 6}
digit*	{LEXVAL : = INSTALL(); return 7}
<	{LEXVAL : = 1; return 8}
<=	{LEXVAL : = 2; return 8}
=	{LEXVAL : = 3; return 8}
<>	{LEXVAL : = 4; return 8}
>	{LEXVAL : = 5; return 8}
>=	{LEXVAL : = 6; return 8}

Fig. 1.26 LEX Program

Q.40. What are the various components of a lexical specification file? Illustrate with an example. (R.G.P.V., June 2011, Dec. 2015, May 2018)

Ans. Refer to Q.39.

Q.41. Develop a lexical analyzer to recognize valid operators of C program. (R.G.P.V., Dec. 2006)

Or

Write a LEX specification file to identify the tokens of the language C. (R.G.P.V., Dec. 2012)

Ans. In the declarations section, declarations are surrounded by the special brackets % {and %}. Anything appearing between these brackets is copied directly into the lexical analyzer, and is not treated as part of the regular definitions or the translation rules. There are two procedures, *install_id* and *install_num*, that are used by the translation rules. Also included in the definitions section are some regular definitions. Each such definition consists of a name and a regular expression denoted by that name.

A lexical analyzer that recognize valid operators of C program is as follows

```
% {
    /* definitions of manifest constants
       LT, LE, EQ, NE, GT, GE,
       IF, THEN, ELSE, ID, NUMBER, RELOP */

    % }
    /* regular definitions */
    delim      { \ t \ n }
    ws          { delim } +
    letter      { A - Z a - z }
    digit       { 0 - 9 }
    id          { letter } ( { letter } | { digit } ) *
    number      { digit } + ( \ { digit } + ) ? ( E [ + | ] ? { digit } + ) ?
    %%

    { ws }      { /* no action and no return */ }
    if          { return (IF); }
    then        { return (THEN); }
    else        { return (ELSE); }
    { id }       { yylval = install_id ( ); return (ID); }
    { number }   { yylval = install_num ( ); return (NUMBER); }
    "<"         { yylval = LT; return (RELOP); }
    "<="       { yylval = LE; return (RELOP); }
    "=="        { yylval = EQ; return (RELOP); }
    "<>"        { yylval = NE; return (RELOP); }
    ">"        { yylval = GT; return (RELOP); }
    ">="       { yylval = GE; return (RELOP); }
    %%
    Install_id ( ) {
        /* procedure to install the lexeme, whose first character is pointed to
           yytext and whose length is yyleng, into the symbol table and return a pointer thereto
        }
        install_num ( ) {
            /* Similar procedure to install a lexeme that is a number */
        }
    }
}
```

Q.42. What is a regular definition? Specify an identifier in C language taking the help of regular definitions. Explain buffer pairs and sentinels

Ans. Refer to Q.35 and Q.41.

Buffer Pair - Refer to Q.33.

(R.G.P.V., May 2011)

Buffer pair code to advance forward (F) pointer is given below --

```
if F at end of first half then begin
    reload second half;
    F := F + 1
end
else if F at end of second half then begin
    reload first half;
    move F to beginning of first half
end
else F := F + 1;
```

Sentinels - If we use the scheme of buffer pairs we must check, each time we advance forward, that we have not moved off one of the buffers, if we do, then we must also reload the other buffer. Thus, for each character read, we make two tests: one for the end of the buffer, and one to determine what character is read (the latter may be a multiway branch).

We can combine the buffer-end test with the test for the current character if we extend each buffer to hold a sentinel character at the end. The sentinel is a special character that cannot be part of the source program, and a natural choice is the character eof. Fig. 1.27 shows the same arrangement as fig. 1.21 but with sentinels added.

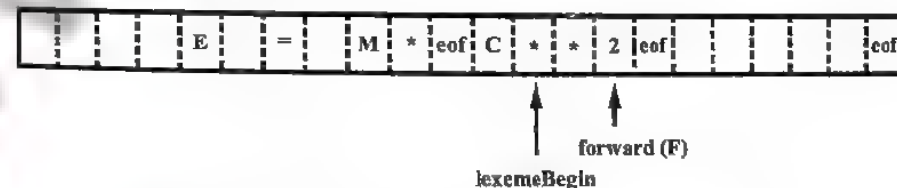


Fig. 1.27 Sentinal at the End of Each Buffer

Look ahead code with sentinels is given below --

```
F := F + 1;
if F = eof then begin
    if F at end of first half then begin
        reload second half;
        F := F + 1
    end
    else if F at the second half then begin
        reload first half;
        move F to beginning of first half
    end
    else terminate lexical analysis
end;
```

NUMERICAL PROBLEMS

Prob.1. Design FA to accept the following –

- (i) Identifiers (ii) Constant.

(R.G.P.V., Dec. 2011)

Sol. (i)

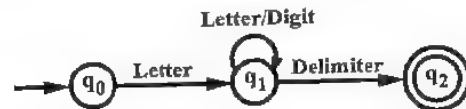


Fig. 1.28 Transition Diagram for Identifier

(ii)

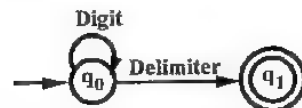


Fig. 1.29 Transition Diagram for Constant

Prob.2. Write a LEX program for the following strings –

w.s. (white space), if then, else, id (R.G.P.V., June 2001)

Or

Write a LEX programme to recognize white space, identifier, if, the and else. (R.G.P.V., Dec. 2001)

Sol. Refer to Q.42.

Prob.3. Write a Lex program to convert infix to postfix.

(R.G.P.V., June 2011)

Sol.

```

%{
#include "y.tab.h"
int yylval;
%}
ID [a-zA-Z]
%%
{ID} {yylval = *yytext; return ID; }
[\n \t] { }
{return *yytext;}
%%
  
```

Prob.4. Write a Lex program to find out total number of vowels and consonants from the given input string. (R.G.P.V., June 2011)

Sol. A lexical analyzer to find out total number of vowels and consonants from the given input string is as follows –

```

%{
/* Definition */
int vowel_cnt = 0, consonant_cnt = 0;
%}

vowel [aeiou] +
consonant [^aeiou]
eol \n
%%
{eol} return 0;
{t} +;
{vowel} {vowel_cnt++;}
{consonant} {consonant_cnt++;}
%%

int main( )
{
printf ("\n Enter some input string \n");
yylex( );
printf ("vowels = %d and consonant = %d \n",
        vowel_cnt, consonant_cnt);
return 0;
}

int yyurap( )
{
return 1;
}

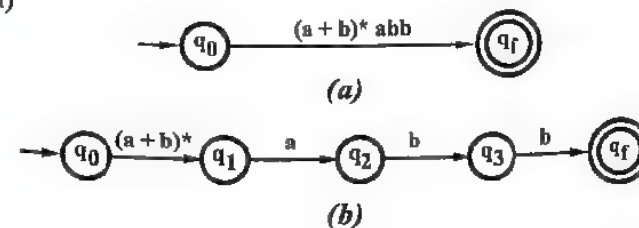
Output –
[root @ app root] # lex vowel.l
[root @ app root] # cc lex.yy.c
[root @ app root] # ./a.out
Enter some input string
John
vowels = 1 and consonants = 3
[root @ app root] #
  
```

Prob.5. Construct FA for the regular expressions –

- (i) $(a + b)^* abb$ (ii) $((a^* + b)^* + b^*)^*$

(R.G.P.V., Dec. 2012)

Sol. (i)



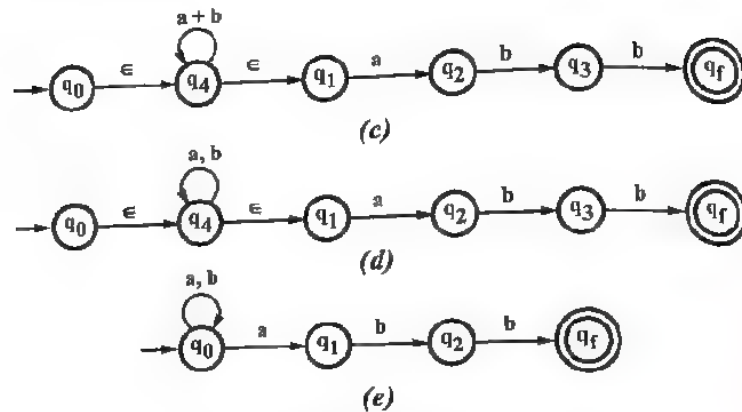


Fig. 1.30 Construction of NFA

First, we construct the NFA with ϵ moves. Then, we eliminate ϵ moves. We start with fig. 1.30 (a). We eliminate the concatenation in the given R.E. by introducing new vertices q_1, q_2 and q_3 and get fig. 1.30 (b).

We eliminate $*$ operation in fig. 1.30 (b) by introducing a new vertex q_4 and ϵ moves as shown in fig. 1.30 (c).

We eliminate $+$ in fig. 1.30 (c) and get fig. 1.30 (d).

We eliminate ϵ moves in fig. 1.30 (d) and get fig. 1.30 (e) which gives the NFA equivalent to the given R.E.

Now, we construct the transition table as shown in table 1.2 corresponding to fig. 1.30.

Table 1.2 Transition Table

State	Input	
	a	b
$\rightarrow q_0$	q_0, q_1	q_0
q_1	—	q_2
q_2	—	q_f
q_f	—	—

Table 1.3

Q	Q_a	Q_b
$\rightarrow [q_0]$	$[q_0q_1]$	$[q_0]$
$[q_0q_1]$	$[q_0q_1]$	$[q_0q_2]$
$[q_0q_2]$	$[q_0q_1]$	$[q_0q_f]$
$[q_0q_f]$	$[q_0q_1]$	$[q_0]$

The successor table is constructed for DFA as shown in table 1.3.

The state diagram for the successor table is the required DFA as shown in fig. 1.31. As q_f is the only final state of NFA, hence $[q_0q_f]$ is the final state of the DFA.

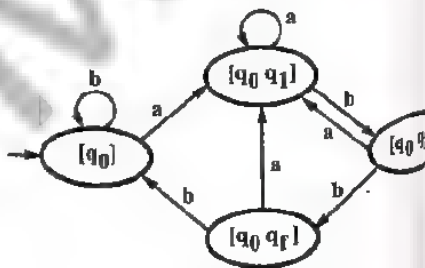


Fig. 1.31 FA

(ii) First, we construct the NFA with ϵ moves for the given regular expression as shown in fig. 1.32 (b)

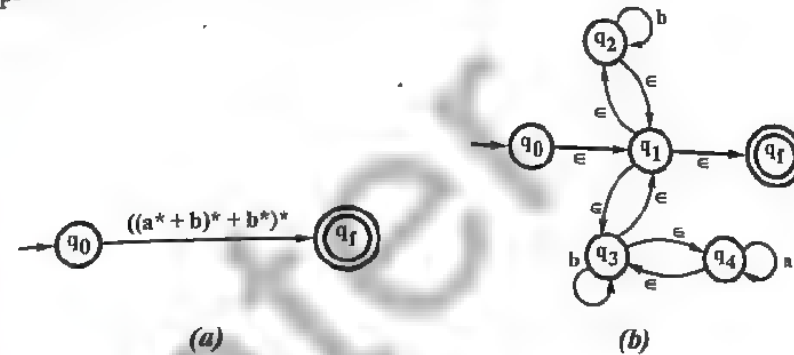


Fig. 1.32

Now, we eliminate all ϵ moves one-by-one to obtain the NFA without ϵ moves as shown in fig. 1.33.

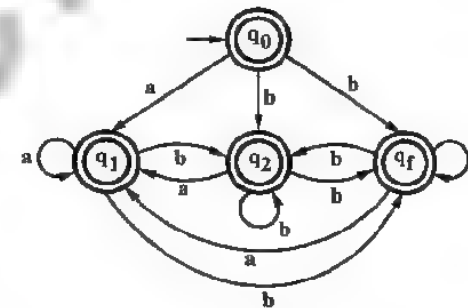


Fig. 1.33

Now, we construct the transition table as shown in table 1.4 corresponding to fig. 1.33.

Table 1.4

State	Input	
	a	b
$\rightarrow q_0$	q_1	q_2, q_f
q_1	q_1	q_2, q_f
q_2	q_1	q_2, q_f
q_f	q_1	q_2, q_f

Table 1.5

Q	Q_a	Q_b
$\rightarrow q_0$	q_1	q_2, q_f
q_1	q_1	q_2, q_f
q_2, q_f	q_1	q_2, q_f

The successor table is constructed for DFA as shown in table 1.5.

The state diagram for the successor table is the required DFA as shown in fig. 1.34.

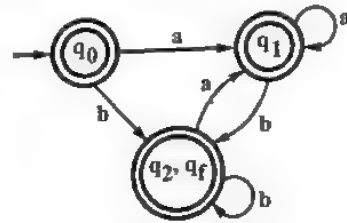


Fig. 1.34 FA

UNIT

2

SYNTAX ANALYSIS AND SYNTAX DIRECTED TRANSLATION

SYNTAX ANALYSIS – CFGs, TOP DOWN PARSING, BRUTE FORCE APPROACH, RECURSIVE DESCENT PARSING

Q.1. Discuss syntax analysis in brief.

Or

What is syntax analysis ? What are its primary functions ?

(R.G.P.V., Dec. 2015)

Ans. The compilation process includes syntax analysis as the next phase to the lexical analysis. Syntax analysis verifies the tokens produced as output a sequence of tokens from lexical analyser are properly sequenced in the accordance with the grammar of the language. The syntax analyzer returns success, if the sequence is defined in the grammar, or failure in case it is not defined. In cases where the statement does not match the grammar specified for the language, the syntax analyser detects the error, emits appropriate error message to the user and if possible, recovers from the error. The data structures used to represent the syntactic structure of a language must now also be recursive rather than linear. The basic structure is usually some kind of tree called a parse tree.

Syntax analysis or parsing is the task of determining the syntax, or structure, of a program. It is performed by a module in the compiler called syntax analyzer or parser. For the parser to perform syntax analysis, the grammar of the language need to be specified. Context-free grammar (CFG) is usually used to define the grammar of a language.

Q.2. What do you understand by a context-free grammar ?

Or

Define context free grammar and explain how it is suitable for parsing.

(R.G.P.V., Dec. 2013)

Or

Give the formal definition of CFG with the help of example.

(R.G.P.V., June 2016)

Ans. Context-free languages are applied in parser design. It is also useful for describing block structure in programming languages.

A grammar $G = (V, T, P, S)$ is said to be context-free grammar (CFG), if every production in P is of the form –

$$A \rightarrow x$$

where, $A \in V$ and $x \in (V \cup T)^*$

A language is said to be context-free iff there is context-free grammar G such that $L = L(G)$.

Every regular grammar is context-free, so a regular language is also context-free. But as we know that there are non-regular languages also. So, we can say that, the family of regular languages are a proper subset of the family of context-free languages.

The name context-free grammar derived from the fact that the substitution of the variable on the left of a production can be made any time such a variable appears in a sentential form. It does not depend on the symbols in the rest of the sentential form. It does not depend on the symbols in the rest of the sentential form (i.e., context). This property is the consequence of allowing only a single variable on the left side of the production. For example – The grammar $G = (\{S\}, \{a, b\}, P, S)$ having production

$$S \rightarrow aSa,$$

$$S \rightarrow bSb,$$

$$S \rightarrow \Lambda,$$

is context-free. A typical derivation in this grammar is as follows –

$$S \Rightarrow aSa \Rightarrow aaSaa \Rightarrow aabSbaa \Rightarrow aabbbaa$$

From this derivation, it is clear that –

$$L(G) = \{u u^R / u \in \{a, b\}^*\}$$

This language is context-free but not regular.

Let us take another example – The grammar G having productions

$$S \rightarrow abB,$$

$$A \rightarrow aaBb,$$

$$B \rightarrow bbAa,$$

$$A \rightarrow \Lambda,$$

is a context-free grammar. Similar to the way given in the above example we can prove that –

$$L(G) = \{ab(bbaa)^n bba(ba)^n / n \geq 0\}$$

Both examples given above involve grammars that are not only context-free, but linear. Regular and linear grammars are clearly context-free, but a context-free grammar is not necessarily linear.

Q.3. What do you mean by context-free grammar? Give distinction between regular and context-free grammar and limitations of context-free grammar. (R.G.P.V., Nov. 2018)

Ans. Refer to Q.2.

To know the differences between regular expressions and context-free grammars, contrast the classic expression grammar against the regular expression as follows –

$$((\text{ident}|\text{num}) (+|-|*|/))^\times (\text{ident}|\text{num})$$

Both the regular expression and the context-free grammar describe the same set of expressions. Consider the notion of a regular grammar to make the difference between regular expressions and context-free grammars clear. Regular grammars have the same expressive power as regular expressions that is, they can describe properly the full set of regular languages.

A regular grammar is a quadruple, $R = (T, NT, S, P)$, with the components having the same meanings as for a context-free grammar. In a regular grammar, however, productions in P are restricted to two forms either $A \rightarrow a$, or $A \rightarrow aB$ where $A, B \in NT$ and $a \in T$.

In contrast, a context-free grammar permits productions with right-hand sides which contain an arbitrary set of symbols from $(T \cup NT)$. Hence, regular grammars are a proper subset of context-free grammars. The same relationship holds for the languages described by regular grammars, called regular languages, and those described by context-free grammars, called context-free languages. The regular languages are a proper subset of the context-free languages.

Limitations of Context-free Grammar – Context-free languages are easy to write and easy to understand and easy to parse with – compared to the more powerful grammars, but they cannot always express what we want. We cannot express english in a context-free grammar, nor we can write a CFG for a programming language. We can work around these problems by defining a CFG for a useful query language that looks like english, or a CFG for a programming language plus a separate type checker.

Generally we can say that CFGs are not perfect, but they are a great tool for a machine.

Q.4. What is a context-free grammar? Illustrate with an example the different components of a context-free grammar. What are the advantages of using CFG to specify a language? (R.G.P.V., May 2018)

Ans. CFG – Refer to Q.2.

Different Components of CFG – A context-free grammar consists of the following components –

(i) A set of terminal symbols, which are the characters of the alphabet that appear in the strings generated by the grammar.

(ii) A set of nonterminal symbols, which are placeholders for patterns of terminal symbols that can be generated by the nonterminal symbols.

(iii) A set of productions, which are rules for replacing (or rewriting) nonterminal symbols (on the left side of the production) in a string with other nonterminal or terminal symbols (on the right side of the production).

(iv) A start symbol, which is a special nonterminal symbol that appears in the initial string generated by the grammar.

Advantages – The Advantages of using CFG to specify a language are as follows –

(i) A grammar provides a precise, yet easy to understand, syntactic specification for the programs of a specific programming language.

(ii) From a properly designed grammar, an efficient parser can be created automatically.

(iii) A grammar imports a structure to a program which is useful for the detection of errors and for its translation into object code.

Q.5. What is parsing ?

Ans. Parsing is the process or technique used to determine whether a string of tokens can be generated by a grammar. A parser must be capable of constructing the tree or the translation is incorrect. A parser can be constructed for any grammar. For any context-free grammar, there is a power that takes at most $O(n^3)$ time to parse a string of n tokens.

Q.6. How do you classify the different parsing techniques ?

(R.G.P.V., Dec. 2011)

Ans. Parsing technique can be classified into two classes –

- (i) Top down approach (ii) Bottom up approach.

These terms refer to the order in which nodes in the parse tree are constructed. In **top down parsing**, construction of parse tree starts at the root and proceeds towards the leaves using the leftmost derivation. While in case of **bottom up parsing**, construction starts at the leaves and proceeds towards the root using reduction process.

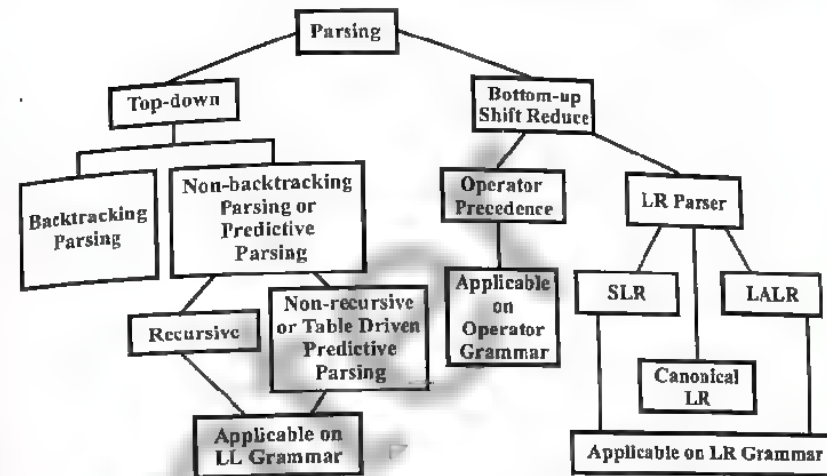


Fig. 2.1 Classification of Parsing Technique

Q.7. What is top-down parsing ? What are the difficulties encountered in this and how are they overcome ?
(R.G.P.V., Dec. 2007, 2010)

Or

Write steps of Brute force approach.

(R.G.P.V., June 2017)

Ans. Top-down parsing attempts to find the left-most derivations for an input string w , which is equivalent to constructing parse tree for the input string w that starts from the root and creates the nodes of the parse tree in a preorder. For example, consider the grammar –

$$S \rightarrow cAd$$

$$A \rightarrow ab \mid a$$

Let the input string be $w = cad$. To construct a parse tree for this sentence top-down, we initially create a tree consisting of a single node labeled S . An input pointer points to c , the first symbol of w . We then use the first production for S to expand the tree and obtain fig. 2.2.



Fig. 2.2 S-production to Expand the Parse Tree

The leftmost leaf, labeled c , matches the first symbol of w , so we now advance the input pointer to a , the second symbol of w , and consider the next leaf, labeled A . We can then expand A using the first alternate for A to obtain the tree shown in fig. 2.3.

The parser now has the match for the second input symbol. We now consider d , the third input symbol, and the next leaf, labeled b . Since b does not match d , we report failure and go back (backtracks) to A , as shown in fig. 2.3. The parser will also reset the input pointer to the second input symbol and it will try a second alternative for A in order to obtain the tree.

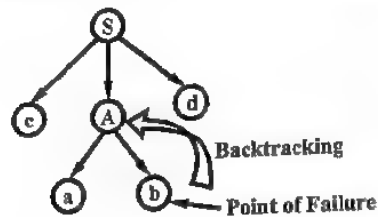


Fig. 2.3

The leaf a matches the second symbol of w and the leaf d matches the third symbol as shown in fig. 2.4. Since we have now produced a parse tree for w, we halt and announce successful completion of parsing.

Difficulties with Top-down Parsing – There are several difficulties with top-down parsing. The first concerns left-recursion. A grammar G is said to

be left-recursive if it has a nonterminal A such that there is a derivation $A \Rightarrow^+ A\alpha$ for some α . A left-recursive grammar can cause a top-down parser to go into an infinite loop. That is, when we try to expand A, we may eventually find ourselves again trying to expand A without having consumed any input. This cycling will surely occur on an erroneous input string, and it may also occur on legal inputs, depending on the order in which the alternates for A are tried. Therefore, to use top-down parsing, we must eliminate all left recursion from the grammar.

A second problem concerns backtracking. If we make a sequence of erroneous expansions and subsequently discover a mismatch, we may have to undo the semantic effects of making these erroneous expansions. For example, entries made in the symbol table might have to be removed. Since undoing semantic actions requires a substantial overhead, it is reasonable to consider top-down parsers that do no backtracking. The recursive descent and predictive parsers are two types of top-down parsers that avoid backtracking.

A third problem with top-down backtracking parsers is that the order in which alternates are tried can affect the language accepted. For example, if the grammar given above we used 'a' and then 'ab' as the order of the alternates for A, we could fail to accept 'cabd'. As shown in fig. 2.4 parse tree and already matched, the failure of the next input symbol, b, to match, would imply that the alternate 'cAd' for S was wrong, leading to rejection of 'cabd'. Therefore, to use top-down parsing, we do left factoring.

Yet another problem is that when failure is reported, we have very little idea where the error actually occurred. In the form given here, a top-down parser with backtrack simply returns failure no matter what the error is.

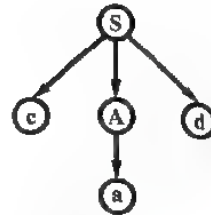


Fig. 2.4

Q.8. Discuss the problems in top-down parsers. How do they overcome?
(R.G.P.V., June 2010)

Ans. Refer to Q.7.

Q.9. What are the causes of backtracking in top-down parser? Explain with an example.
(R.G.P.V., Dec. 2017)

Ans. Refer to Q.7.

Q.10. Write short note on recursive descent parser.

Ans. Recursive descent parsing is a top-down method of syntax analysis in which we execute a set of procedures to process the input. In many practical cases a top-down parser needs no backtrack. In order that no backtracking be required, we must know, given the current input symbol a and the nonterminal A to be expanded, which one of the alternates of production $A \rightarrow \alpha_1 | \alpha_2 | \dots | \alpha_n$ is the unique alternate that derives a string beginning with a. That is, the proper alternate is detectable by looking at only the first symbol it derives. For example, control constructs, with their distinguishing keywords, are detectable in this way. Suppose we have productions –

Statement \rightarrow

if condition then statement else statement

while condition do statement

begin statement-list end

Then the keywords **if**, **while**, and **begin** tell us which alternate is the only one that could possibly succeed if we are to find a statement.

One nuance concerns the empty string. If one alternate for A is ϵ , and none of the other alternates derives a string beginning with a, then on input a we may expand A by $A \rightarrow \epsilon$, that is, we succeed without further a do in recognizing an A.

A parser that uses a set of recursive procedures to recognize its input with no backtracking is called a *recursive-descent parser*. The recursive procedures can be quite easy to write and fairly efficient if written in a language that implements procedure calls efficiently. To avoid the necessity of a recursive language, we shall also consider a tabular implementation of recursive descent, called predictive parsing, where a stack is maintained by the parser, rather than by the language in which the parser is written.

NUMERICAL PROBLEMS

Prob.1. Consider a grammar $G, S \rightarrow SaS|b$, show that G is ambiguous for string 'bababab'.
(R.G.P.V., June 2010)

Sol. If the given grammar G for string 'bababab' has two derivation trees, then, the grammar is ambiguous. The derivation or parse are given

below –

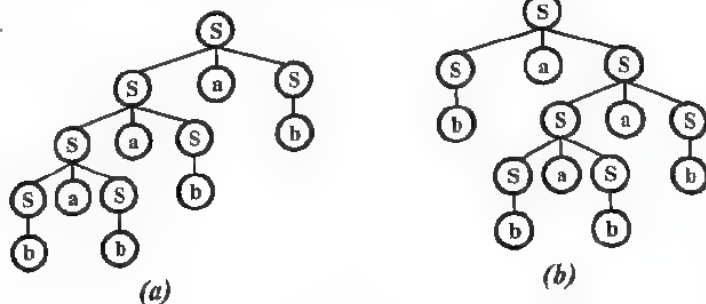


Fig. 2.5

Prob.2. Remove null production, unit production and useless symbols from the following CFG –

$$S \rightarrow PQ \mid RS \mid QT$$

$$P \rightarrow a \mid aP$$

$$Q \rightarrow \varepsilon \mid bQ$$

$$R \rightarrow Rd \mid Rc$$

$$T \rightarrow eT \mid \varepsilon$$

(R.G.P.V., Dec. 2008)

Sol. We find the nullable variables and then construct the CFG without ε -productions in the following manner –

To eliminate $Q \rightarrow \varepsilon$ from this grammar, the non ε -productions to be added are obtained as follows – the list of the productions containing Q on the right-hand side is –

$$S \rightarrow PQ \mid QT$$

$$Q \rightarrow bQ \mid \varepsilon$$

Replace each occurrence of Q to obtain the non- ε -productions. The list of these productions is –

$$S \rightarrow P \mid T$$

$$Q \rightarrow b$$

Add these productions to the grammar, and eliminate $Q \rightarrow \varepsilon$ from the grammar. This gives us the following grammar –

$$S \rightarrow PQ \mid RS \mid QT \mid P \mid T$$

$$P \rightarrow a \mid aP$$

$$Q \rightarrow bQ \mid b$$

$$R \rightarrow Rd \mid Rc$$

$$T \rightarrow eT \mid \varepsilon$$

Similarly to eliminate $T \rightarrow \varepsilon$, we get

$$S \rightarrow PQ \mid RS \mid QT \mid Q \mid P \mid T$$

$$P \rightarrow a \mid aP$$

$$Q \rightarrow bQ \mid b$$

$$R \rightarrow Rd \mid Rc$$

$$T \rightarrow eT \mid e$$

The given grammar contains the productions –

$$S \rightarrow Q$$

$$S \rightarrow T$$

$$S \rightarrow P$$

which are the unit productions. To eliminate these productions from the given grammar, we first select the unit production $S \rightarrow Q$. There exists a non-unit Q production which is $Q \rightarrow b$. Hence, we add $S \rightarrow b$ to the grammar and eliminate $S \rightarrow Q$.

Similarly $S \rightarrow e$ is added and eliminate the $S \rightarrow T$.

Further, add the $S \rightarrow a$ and eliminate the $S \rightarrow P$.

Therefore, the grammar that we get that does not contain unit productions is

$$S \rightarrow PQ \mid RS \mid QT \mid a \mid b \mid e$$

$$P \rightarrow a \mid aP$$

$$Q \rightarrow bQ \mid b$$

$$R \rightarrow Rd \mid Rc$$

$$T \rightarrow eT \mid e$$

Since $P \rightarrow a$, $Q \rightarrow b$, $T \rightarrow e$ are the productions of the form $A \rightarrow w$, where w is in T^* , non-terminal P , Q , T are capable of deriving to w to T^* .

There are productions $S \rightarrow PQ \mid QT$ and $P \rightarrow aP$, $Q \rightarrow bQ$, $T \rightarrow eT$.

The right side of which contains, non-terminals P , Q and T , all of them which are known to derivable to w in T^* , and S is also capable of deriving to w in T^* .

There are three productions $S \rightarrow RS$, $R \rightarrow Rd$, $R \rightarrow Rc$, the right side of these productions contain non-terminals R , S which are known to non-derivable to w in T^* . Hence non-terminal R , S is not capable of deriving to w in T^* . Therefore, by eliminating the productions containing R and S . We get useless symbols –

$$S \rightarrow PQ \mid QT \mid a \mid b \mid e$$

$$P \rightarrow a \mid aP$$

$$Q \rightarrow bQ \mid b$$

$$T \rightarrow eT \mid e$$

Prob.3. Consider the grammar –

$$S \rightarrow (L)a$$

$$L \rightarrow L, S \mid S$$

(i) Find Parse tree for the following –

(A) (a, a)

(B) $(a, (a, a))$

(C) $(a, ((a, a), (a, a)))$

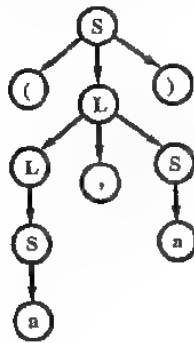
(ii) Construct leftmost derivative for each (a).

(iii) Construct rightmost derivative for each (a).

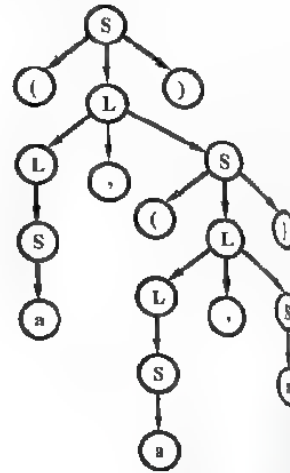
(R.G.P.V., June 2005, 2006)

Sol. (i) Parse trees for each expression are given in fig. 2.6.

(A) Parse tree for (a, a) – (B) Parse tree for (a, (a, a)) –

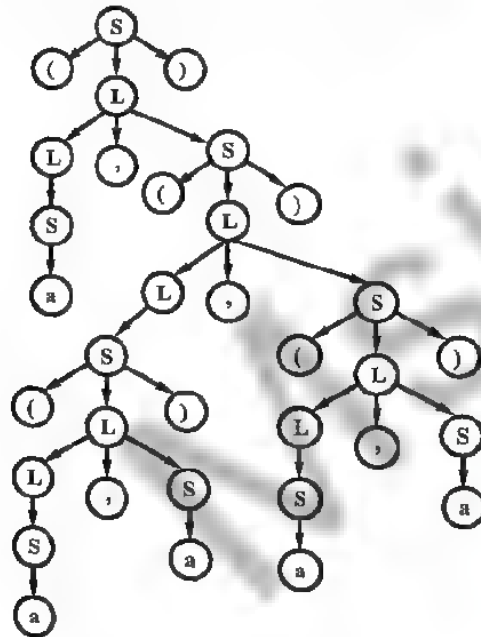


(a)



(b)

(C) Parse tree for (a, ((a, a), (a, a))) –



(c)

Fig. 2.6 Parse Trees

(ii) The leftmost derivation for each (a) as –

(A) (a, a)

$S \rightarrow (L) \rightarrow (L, S) \rightarrow (S, S) \rightarrow (a, S) \rightarrow (a, a)$

(B) (a, (a, a))

$S \rightarrow (L) \rightarrow (L, S) \rightarrow (S, S) \rightarrow (a, S) \rightarrow (a, (L)) \rightarrow (a, (L, S)) \rightarrow (a, (S, S)) \rightarrow (a, (a, S)) \rightarrow (a, (a, a))$

(C) (a, ((a, a), (a, a)))

$S \rightarrow (L) \rightarrow (L, S) \rightarrow (S, S) \rightarrow (a, S) \rightarrow (a, (L)) \rightarrow (a, (L, S)) \rightarrow (a, ((L, S), S)) \rightarrow (a, ((S, S), S)) \rightarrow (a, ((a, S), S)) \rightarrow (a, ((a, a), S)) \rightarrow (a, ((a, a), (L))) \rightarrow (a, ((a, a), (L, S))) \rightarrow (a, ((a, a), (S, S))) \rightarrow (a, ((a, a), (a, S))) \rightarrow (a, ((a, a), (a, a)))$

(iii) Rightmost derivation –

(A) (a, a)

$S \rightarrow (L) \rightarrow (L, S) \rightarrow (L, a) \rightarrow (S, a) \rightarrow (a, a)$

(B) (a, (a, a))

$S \rightarrow (L) \rightarrow (L, S) \rightarrow (L, (L)) \rightarrow (L, (L, S)) \rightarrow (L, (L, a)) \rightarrow (L, (S, a)) \rightarrow (L, (a, a)) \rightarrow (S, (a, a)) \rightarrow (a, (a, a))$

(C) (a, ((a, a), (a, a)))

$S \rightarrow (L) \rightarrow (L, S) \rightarrow (L, (L)) \rightarrow (L, (L, S)) \rightarrow (L, (L, (L))) \rightarrow (L, (L, (L, S))) \rightarrow (L, (L, (L, a))) \rightarrow (L, (L, (S, a))) \rightarrow (L, (L, (S, a)))$

(iii) The parse tree –

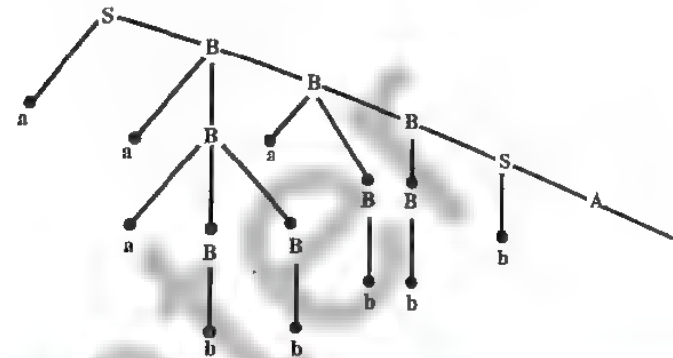


Fig. 2.7 Parse Tree for 'aaabbabbbba'

$$S \rightarrow aB \mid bA$$
$$A \rightarrow a \mid aS \mid bAA$$
$$B \rightarrow b \mid bS \mid aBB$$

TRANSFORMATION ON THE GRAMMARS, PREDICTIVE PARSING, BOTTOM UP PARSING

(i) Leftmost derivation (ii) Rightmost derivation (iii) Parse tree.
(R.G.P.V., Dec. 2006)

Q.11. What kind of transformations done to the grammar rules to suit a predictive parser ?

$$S \rightarrow aB$$

→ aaBB

→ aaaBBB

→ aaabBB

→ aaabbB

→ aaabbaBB

→ aaabbabB

→ aaabbabbS

→ aaabbabbbA

→ aaabbabbba

(ii) Rightmost derivation for string **aaabbabbbba** is as follows-

$$S \rightarrow aB$$

→ aaBB

→ aaBbS

→ aaBbbA

→ aaBbba

→ aa**B**Bbba

→ aaabBbSbba

→ aaaBbaBbba

→ aaabbbba

→ aaabbabbba

Ans. There are two types of transformations done to the grammar in order to suit a predictive parser. They are as follows –

(i) **Elimination of Left Recursion** – A grammar is left recursive if it has a nonterminal A such that there is a derivation $A \xRightarrow{+} A\alpha$ for some string α . Top-down parsing methods cannot handle left-recursive grammars, so a transformation that eliminates left recursion is needed. In general left-recursive pair of productions $A \rightarrow A\alpha|\beta$ could be replaced by the non-left-recursive productions

$$A \rightarrow \beta A'$$

$$A \rightarrow \alpha A' \mid \epsilon$$

without changing the set of strings derivable from A .

(ii) **Left Factoring** – Left-factoring is a grammar transformation that is useful for producing a grammar suitable for predictive parsing. The basic idea is that when it is not clear which of two alternative productions to use to expand a nonterminal A, we may be able to rewrite the A-productions to defer the decision until we have seen enough of the input to make the right choice.

In general, if $A \rightarrow \alpha \beta_1 | \alpha \beta_2$ are two productions, and the input begins with a nonempty string derived from α , we do not know whether to expand A to $\alpha \beta_1$ or to $\alpha \beta_2$. However, we may defer the decision by expanding

A to $\alpha A'$ and A' to β_1 , or to β_2 . That is, left-factored, the original productions become,

$$\begin{aligned} A &\rightarrow \alpha A' \\ A' &\rightarrow \beta_1 | \beta_2 \end{aligned}$$

Q.12. How do you classify the different parsing techniques? Define left recursion and left factoring. Consider the following grammar.

$$\begin{aligned} A &\rightarrow ABd | Aa | a \\ B &\rightarrow Be | b \end{aligned}$$

remove left recursion.

(R.G.P.V., May 2013)

Ans. Refer to Q.6, Q.11 and prob.5.

Q.13. Explain left recursion and show how it is eliminated. Describe the algorithm used for eliminating left recursion.

(R.G.P.V., Dec. 2013)

Ans. Refer to Q.11 (i).

Algorithm –

- (i) Arrange the nonterminals in some order N_1, N_2, \dots, N_n .
- (ii) for $i := 0$ to $n - 1$ do begin
 - for $j := 0$ to $i - 2$ do begin
 - replace each production of the form $N_i \rightarrow N_j \gamma$ by the productions $N_i \rightarrow \delta_1 \gamma | \delta_2 \gamma | \dots | \delta_k \gamma$, where $N_j \rightarrow \delta_1 | \delta_2 | \dots | \delta_k$ are all the current N_j - productions;
- end
- eliminate the immediate left recursion among the N_i productions
- end

Q.14. Define – Left recursive. State the rule to remove left recursion from the grammar. Eliminate left recursive from following grammar.

$$\begin{aligned} S &\rightarrow Aa | b \\ A &\rightarrow Ac | Sd | f \end{aligned}$$

(R.G.P.V., Dec. 2014)

Ans. Refer to Q.11.

Eliminating the immediate left recursion (productions of the form $A \rightarrow A\alpha$ to the production A could be replaced by the non-left-recursive productions.

$$\begin{aligned} A &\rightarrow \beta A' \\ A' &\rightarrow \alpha A' | \epsilon \end{aligned}$$

without changing the set of strings derivable from A

$$\begin{aligned} S &\rightarrow Aa | b \\ A &\rightarrow SdA' | fA' \\ A' &\rightarrow cA' | \epsilon \end{aligned}$$

Q.15. What is predictive parser? How a parser is controlled by a program?

(R.G.P.V., Dec. 2003, 2007, 2010)

Ans. A predictive parser is an efficient way of implementing recursive-descent parsing by handling the stack of activation records explicitly. A predictive parser is shown in fig. 2.8.

The predictive parser has an input, a stack, a parsing table, and an output. The input contains the string to be parsed, followed by \$, the right endmarker. The stack contains a sequence of grammar symbols, preceded by \$, the bottom-of-stack marker. Initially, the stack contains the start symbol of the grammar preceded by \$. The parsing table is a two dimensional array $M[A, a]$, where A is a nonterminal, and 'a' is a terminal or the symbol \$.

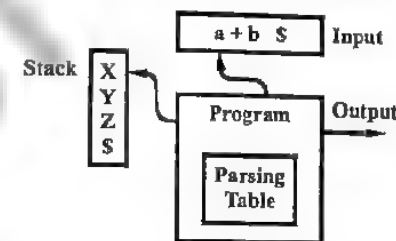


Fig. 2.8 Model of a Predictive Parser

The parser is controlled by a program that behaves as follows. The program determines X , the symbol on top of the stack, and 'a', the current input symbol. These two symbols determine the action of the parser. There are three possibilities –

- (i) If $X = a = \$$, the parser halts and announces successful completion of parsing.
- (ii) If $X = a \neq \$$, the parser pops X off the stack and advances the input pointer to the next input symbol.
- (iii) If X is a nonterminal, the program consults entry $M[X, a]$ of the parsing table M . This entry will be either an X -production of the grammar or an error entry. If $M[X, a] = \{X \rightarrow UVW\}$, the parser replaces X on top of the stack by WVU . As output, the grammar does the semantic action associated with this production, which, for the time being, we shall assume is just printing the production used. If $M[X, a] = \text{error}$, the parser calls an error-recovery routine.

Consider the grammar

$$\begin{aligned} E &\rightarrow TE' \\ E' &\rightarrow + TE' | \epsilon \\ T &\rightarrow FT' \\ T' &\rightarrow * FT' | \epsilon \\ F &\rightarrow (E) | id \end{aligned}$$

A predictive parsing table for this grammar is shown in table 2.1. Blanks are error routines.

Table 2.1 Parsing Table M for Grammar

Non-terminal	Input Symbol					
	id	+	*	()	\$
E	$E \rightarrow TE'$	$E' \rightarrow +TE'$		$E \rightarrow TE'$	$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
E'	$T \rightarrow FT'$			$T \rightarrow FT'$		
T		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$	$F \rightarrow (E)$	$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
T'						
F	$F \rightarrow id$					

With input $id + id * id$ the parser works in the following way by making use of given parsing table.

Table 2.2 Moves Made by Predictive Parser on Input $id + id * id$

Stack	Input	Output
SE	$id + id * id \$$	
SE'T	$id + id * id \$$	$E \rightarrow TE'$
SE'T'F	$id + id * id \$$	$T \rightarrow FT'$
SE'T'id	$id + id * id \$$	$F \rightarrow id$
SE'T'	$+ id * id \$$	
SE'	$+ id * id \$$	$T' \rightarrow \epsilon$
SE'T +	$+ id * id \$$	$E' \rightarrow +TE'$
SE'T	$id * id \$$	
SE'T'F	$id * id \$$	$T \rightarrow FT'$
SE'T'id	$id * id \$$	$F \rightarrow id$
SE'T'	$* id \$$	
SE'T'F*	$* id \$$	$T' \rightarrow *FT'$
SE'T'F	$id \$$	
SE'T'id	$id \$$	$F \rightarrow id$
SE'T'	$\$$	
SE'	$\$$	$T' \rightarrow \epsilon$
\$	$\$$	$E' \rightarrow \epsilon$

FIRST and FOLLOW – To construct a parsing table for predictive parsing two functions are used which are called **FIRST** and **FOLLOW**. These functions allow to fill in the entries of predictive parsing table.

FIRST(α) is the function applied on a string of variable and terminal any size is defined as the set of the terminals that begin the strings derived from α . If $\alpha \Rightarrow \epsilon$ Then ϵ is also in **FIRST(α)**.

To compute **FIRST(X)** the following rules are applied –

- If X is terminal, then **FIRST(X)** is X .
- If $X \rightarrow \epsilon$ is a production, then add ϵ to **FIRST(X)**.
- If X is a nonterminal and $X \rightarrow Y_1 Y_2 \dots Y_k$ is any production place 'a' in **FIRST(X)** if 'a' is in **FIRST(Y_i)** and ϵ is in all **FIRST(Y_j)** for $j = 1, 2, \dots, k$. If ϵ is in all **FIRST(Y_j)** where $j = 1, 2, \dots, k$ then add ϵ to **FIRST(X)**.

FOLLOW(A) is the function applied on a nonterminal and defined as the set of terminals that can appear immediately to the right of A .

To compute **FOLLOW(A)** for all nonterminals A , apply the following rules –

- Place $\$$ in **FOLLOW(S)**, where S is the start symbol.
- If there is a production $A \rightarrow \alpha B \beta$ then everything in **FIRST(β)** except for ϵ is placed in **FOLLOW(B)**.
- If there is a production $A \rightarrow \alpha B$ or a production $A \rightarrow \alpha B \beta$ where **FIRST(β)** contains ϵ (i.e., $\beta \Rightarrow^* \epsilon$), then everything in **FOLLOW(A)** is in **FOLLOW(B)**.

By making use of above discussed grammar which is unambiguous, left recursion free, **FIRST** and **FOLLOW** function are determined.

FIRST(E) = FIRST(T) = FIRST(F) = { (, id }

FIRST(E') = { +, ϵ }

FIRST(T') = { *, ϵ }

FOLLOW(E) = FOLLOW(E') = {), \$ }

FOLLOW(T) = FOLLOW(T') = { +,), \$ }

FOLLOW(F) = { +, *,), \$ }

Construction of Parsing Table – The idea behind the algorithm is the following. If the **FIRST** and **FOLLOW** functions are evaluated then this algorithm is applied. If $A \rightarrow \alpha$ is a production with a in **FIRST(α)**, then the parser will expand A by α when current the input symbol is a . The only complication occurs $\alpha = \epsilon$ or $\alpha \Rightarrow^* \epsilon$ then we should expand A by α if the current input symbol is in **FOLLOW(A)** or if the $\$$ on the input has been reached and $\$$ is in **FOLLOW(A)**.

ALGORITHM

- For each production $A \rightarrow \alpha$ of the grammar do steps (ii) and (iii).
- For each terminal a in **FIRST(α)** add $A \rightarrow \alpha$ to $M[A, a]$.
- If ϵ is in **FIRST(α)**, add $A \rightarrow \alpha$ to $M[A, b]$ for each terminal b in **FOLLOW(A)**. If ϵ is in **FIRST(α)** and $\$$ is in **FOLLOW(A)** add $A \rightarrow \alpha$ to $M[A, \$]$.
- Make each undefined entry of M be error.

Q.16. What are the components of a table-driven predictive parser ? Write table-driven predictive parsing algorithm. (R.G.P.V., May 2019)

Ans. Refer to Q.15.

Q.17. Show whether the following grammar is LL(1) or not

$E \rightarrow TE/+TE/\epsilon$

$T \rightarrow FT/*FT/\epsilon$

$F \rightarrow (E)/id$

And explain the model of predictive parser. (R.G.P.V., Dec. 2016)

Ans. Refer to Q.15.

Q.18. Write the advantages of table driven predictive parsing.

Ans. Some advantages are as follows –

- (i) It is easy to generate a table-driven parser from a given grammar. The parsing program is independent of the grammar and remains common to any grammar. The parsing table is the only component that depends on the grammar and can be generated by using the FIRST and FOLLOW set generation algorithms.
- (ii) The error recovery and reporting in table-driven parser can be done easily by having entries in the table, which point to the error recovery and reporting routines.

Q.19. Discuss the implementation of shift-reduce parsing.

Ans. A convenient way to implement a shift-reduce parser is to use a stack and an input buffer. The bottom of the stack and the right end of the input is marked by \$.

Stack	Input
\$	w\$

The parser operates by shifting zero or more input symbols onto the stack until a handle β is on top of the stack. The parser then reduces β to the left side of the appropriate production. The parser repeats this cycle until it has detected an error or until the stack contains the start symbol and the input is empty –

Stack	Input
\$\$	\$

At this position, the parser halts and announce successful completion of parsing.

The sequence of actions a shift-reduce parser might make in parsing the input string $id_1 + id_2 * id_3$ according to the grammar –

$E \rightarrow E + E$
 $E \rightarrow E * E$
 $E \rightarrow (E)$
 $E \rightarrow id$

using the rightmost derivation is given in table 2.3.

Table 2.3 Shift-reduce Parsing Actions

S.No.	Stack	Input	Action
(i)	\$	$id_1 + id_2 * id_3 \$$	shift
(ii)	$\$id_1$	$+ id_2 * id_3 \$$	reduce by $E \rightarrow id$
(iii)	$\$E$	$+ id_2 * id_3 \$$	shift
(iv)	$\$E +$	$id_2 * id_3 \$$	shift
(v)	$\$E + id_2$	$* id_3 \$$	reduce by $E \rightarrow id$
(vi)	$\$E + E$	$* id_3 \$$	shift
(vii)	$\$E + E *$	$id_3 \$$	shift
(viii)	$\$E + E * id_3$	\$	reduce by $E \rightarrow id$
(ix)	$\$E + E * E$	\$	reduce by $E \rightarrow E * E$
(x)	$\$E + E$	\$	reduce by $E \rightarrow E + E$
(xi)	$\$E$	\$	accept

While the primary operations of the parser are shift and reduce, there are actually four possible actions a shift-reduce parser can make –

- (i) **Shift** – The next input symbol is shifted to the top of the stack.
- (ii) **Reduce** – The parser knows the right end of the handle is at the top of the stack. It must then locate the left end of the handle within the stack and decide with what nonterminal to replace the handle.
- (iii) **Accept** – The parser announces successful completion of parsing.
- (iv) **Error** – The parser discovers that a syntax error has occurred and calls an error recovery routine.

Q.20. Describe the conflicts that may occur during shift reduce parsing. (R.G.P.V., Dec. 2014)

Ans. There are context-free grammars for which shift-reduce parsing cannot be used. Every shift-reduce parser for such a grammar can reach a configuration in which the parser, knowing the entire stack contents and the next input symbol, cannot decide whether to shift or to reduce (a shift/reduce conflict), or cannot decide which of several reductions to make (a reduce/reduce conflict).

Q.21. Discuss bottom-up parsing in brief.

Ans. A bottom-up parser uses an explicit stack to perform a parse, similar to a non-recursive top-down parser. The parsing stack will contain both tokens and nonterminals and also some extra state information. Initially the stack is empty and will contain the start symbol at the end of a successful parse. A schematic for bottom-up parsing is, therefore,

\$	InputString \$
...	...
...	...
\$ StartSymbol	\$ accept

where the parsing stack is on the left, the input is in the center and the actions of the parser are on the right.

A bottom-up parser has two possible actions (besides “accept”) –

(i) **Shift** – A terminal from the front of the input is shifted to the top of the stack.

(ii) **Reduce** – Used to reduce a string α at the top of the stack to a nonterminal A, given the BNF choice $A \rightarrow \alpha$.

A bottom-up parser is thus sometimes called a **shift-reduce parser**. Shift actions are indicated by writing the word shift. Reduce actions are indicated by writing the word reduce and giving the BNF choice used in the reduction. In bottom-up parsers grammars are always augmented with a new start symbol. This means that if S is the start symbol, a new start symbol S' is added to the grammar, with a single unit production to the previous start symbol –

$S' \rightarrow S$

Bottom-up parsers have less difficulty than top-down parsers with lookahead. Indeed, a bottom-up parser can shift input symbols onto the stack until it determines what actions to perform. However, a bottom-up parser may need to look deeper into the stack than just the top in order to determine what action to perform.

Q.22. Discuss error recovery in predictive parsing. Explain shift reduce parsing with taking suitable examples. (R.G.P.V., May 2019)

Ans. The stack of a nonrecursive predictive parser makes explicit the terminals and nonterminals that the parser hopes to match with the remainder of the input. An error is detected during predictive parsing when the terminal on top of the stack does not match the next input symbol or when nonterminal B is on top of the stack, b is the next input symbol, and the parsing table entry $M[B, b]$ is empty.

Panic-mode error recovery is based on the idea of skipping symbols on the input until a token in a selected set of synchronizing tokens appears. Its effectiveness depends on the choice of synchronizing set. The sets should be chosen so that the parser recovers quickly from errors that are likely to occur in practice. Some points are as follows –

(i) As a starting point, we can place all symbols in FOLLOW(B) into the synchronizing set for nonterminal B. If we skip tokens until an element of FOLLOW(B) is seen and pop B from the stack, it is likely that parsing can continue.

(ii) It is not enough to use FOLLOW(B) as the synchronizing set for B. For example, if semicolons terminate statements, as in C, then keywords that begin statements may not appear in the FOLLOW set of the nonterminal generating expressions. A missing semicolon after an assignment may therefore result in the keyword beginning the next statement being skipped. Often, there is a hierarchical structure on constructs in a language; e.g., expressions appear within statement, which appear within blocks and so on. We can add to the synchronizing set of a lower construct the symbols that begin higher constructs. For example, we might add keywords that begin statements to the synchronizing sets for the nonterminals generating expressions.

(iii) If we add symbols in FIRST(B) to the synchronizing set for nonterminal B, then it may be possible to resume parsing according to B if a symbol in FIRST(B) appears in the input.

(iv) If a nonterminal can generate the empty string, then a production deriving ϵ can be used as a default. Doing so may postpone some error detection, but cannot cause an error to be missed. This approach reduces the number of nonterminals that have to be considered during error recovery.

(v) If a terminal on top of the stack cannot be matched, a simple idea is to pop the terminal, issue a message saying that the terminal was inserted

and continue parsing. In effect, this approach takes the synchronizing set of a token to consist of all other tokens.

Refer to Q.21 and prob.8.

Q.23. Distinguish between top-down parsing and bottom-up parsing. What is the largest class of grammars that can be parsed by each of them? (R.G.P.V., May 2018)

Ans.

S.No.	Top Down Parsing	Bottom Up Parsing
(i)	This parsing strategy first looks at the highest level of the parse tree and works down the parse tree by using the rules of a formal grammar.	This parsing strategy first looks at the lowest level of the parse tree and works up the parse tree by using the rules of a formal grammar.
(ii)	The parsing occurs from the starting symbol to the input string.	The parsing occurs from the input string to the starting symbol.
(iii)	Employed to select what production rule to use in order to construct the string.	Employed to select when to use a production rule to reduce the string to get the starting symbol.
(iv)	Top down parsing uses left most derivation.	Bottom up parsing uses right most derivation.

From applicability point of view, the largest class of grammars for which top-down parsers can work successfully is known as LL grammars. The largest class of grammars for which bottom up parsers can succeed is known as LR grammars.

Q.24. Explain handle pruning. Explain the same for the grammar $E \rightarrow E + E / E * E / (E) / id$ and input string $id1 + id2 * id3$.

(R.G.P.V., Dec. 2014)

Ans. A handle of a right-sentential form γ is a production $A \rightarrow \beta$ and a position of γ where the string β may be found and replaced by A to produce the previous right-sentential form in a rightmost derivation of γ . That is, if $S \xrightarrow{rm} \alpha A w \xrightarrow{rm} \alpha \beta w$, then $A \rightarrow \beta$ in the position following α is a handle of $\alpha \beta w$. The string w to the right of the handle contains only terminal symbols. In other words, we say “the substring β is a handle of $\alpha \beta w$ ” if the position of β and the production $A \rightarrow \beta$ we have in mind are clear. If a grammar is unambiguous, then every right-sentential form of the grammar has exactly one handle.

A rightmost derivation in reverse, often called a canonical reduction sequence, is obtained by “handle pruning”. That is, we start with a string of terminals w which we wish to parse. If w is a sentence of the grammar at

hand, then $w = \gamma_n$, where γ_n is the n th right-sentential form of some as yet unknown rightmost derivation

$$S = \gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \dots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n = w$$

To reconstruct this derivation in reverse order, we locate the handle β_n in γ_n and replace β_n by the left side of some production $A_n \rightarrow \beta_n$ to obtain the $(n-1)$ st right-sentential form γ_{n-1} . We then repeat this process. That is, we locate the handle β_{n-1} in γ_{n-1} and reduce this handle to obtain the right-sentential form γ_{n-2} . If by continuing this process we produce a right-sentential form consisting only of the start symbol S , then we halt and announce successful completion of parsing. The reverse of the sequence of productions used in the reductions is a rightmost derivation for the input string.

The sequence of steps for the input string $\text{id1} + \text{id2} * \text{id3}$ according to the given grammar using handle pruning is given in table 2.4.

Table 2.4

S.No.	Input	Handle	Production Used
(i)	$E + \text{id2} * \text{id3}$	id1	$E \rightarrow \text{id}$
(ii)	$E + E * \text{id3}$	id2	$E \rightarrow \text{id}$
(iii)	$E + E * E$	id3	$E \rightarrow \text{id}$
(iv)	$E + E$	$E * E$	$E \rightarrow E * E$
(v)	E	$E + E$	$E \rightarrow E + E$

NUMERICAL PROBLEMS

Prob.5. Consider the following grammar –

$$A \rightarrow ABd|Aa|a$$

$$B \rightarrow Be|b$$

remove left recursion from above grammar.

(R.G.P.V., Dec. 2001)

Sol. Eliminating the immediate left recursion (productions of the form $A \rightarrow A\alpha$) to the production A and then B could be replaced by the non-left recursive productions

$$A \rightarrow \beta A'$$

$$A' \rightarrow \alpha A' | \epsilon$$

without changing the set of strings derivable from A .

$$A \rightarrow aA'$$

$$A' \rightarrow BdA'|aA'| \epsilon$$

$$B \rightarrow bB'$$

$$B' \rightarrow eB'| \epsilon$$

Prob.6. Remove left recursion from the following CFG –

$$S \rightarrow SSa|Ab|c$$

$$A \rightarrow AAb|d$$

(R.G.P.V., Dec. 2004)

Sol. Eliminating the immediate left recursion (productions of the form $A \rightarrow A\alpha$) to the production S and then A could be replaced by the non-left-recursive productions

$$A \rightarrow \beta A'$$

$$A' \rightarrow \alpha A' | \epsilon$$

without changing the set of strings derivable from A .

$$S \rightarrow AbS'|cS'$$

$$S' \rightarrow SaS'| \epsilon$$

$$A \rightarrow dA'$$

$$A' \rightarrow AbA'| \epsilon$$

Prob.7. Do left factoring the following –

$$S \rightarrow aS|aA|b$$

$$A \rightarrow aAb|aA|a$$

(R.G.P.V., Dec. 2004)

Sol. In the given grammar, apply left factoring procedure –

$$A \rightarrow \alpha \beta_1 | \alpha \beta_2$$

$$A \rightarrow \alpha A'$$

$$A' \rightarrow \beta_1 | \beta_2$$

Then, new productions are –

$$S \rightarrow aS'|b$$

$$S' \rightarrow S|A$$

$$A \rightarrow aAA'|a$$

$$A' \rightarrow b| \epsilon$$

Prob.8. Design shift reduce parser for the following grammar –

$$S \rightarrow ISI | OSO | 2$$

(R.G.P.V., June 2004)

Sol. The sequence of steps through the actions a shift-reduce parser might make in parsing the input string 10201 according to grammar, using the right most derivation is given in table 2.5.

Table 2.5 Configurations of Shift-reduce Parser

S.No.	Stack	Input	Action
(i)	\$	10201\$	shift
(ii)	\$1	0201\$	shift
(iii)	\$10	201\$	shift
(iv)	\$102	01\$	reduce by $S \rightarrow 2$

(v)	\$10S	01\$	shift
(vi)	\$ 10S0	1\$	reduce by $S \rightarrow CS0$
(vii)	\$1S	1\$	shift
(viii)	\$1S1	\$	reduce by $S \rightarrow 1S1$
(ix)	\$S	\$	accept

Prob.9. Consider the grammar

$$S \rightarrow ACB|CbB|Ba$$

$$A \rightarrow da|BC$$

$$B \rightarrow g|\epsilon$$

$$C \rightarrow h|\epsilon$$

Calculate FIRST and FOLLOW.

(R.G.P.V., June 2009, 2010, 2011, Dec. 2011, 2012)

Sol. Computation of FIRST –

$$\text{FIRST}(S) = \text{FIRST}(ACB) \cup \text{FIRST}(CbB) \cup \text{FIRST}(Ba)$$

$$= \{(d, g, h, \epsilon) \cup (h, b) \cup (g, a)\}$$

$$= \{d, g, h, \epsilon, b, a\}$$

$$\text{FIRST}(A) = \text{FIRST}(da) \cup \text{FIRST}(BC)$$

$$= \{(d) \cup (g, h, \epsilon)\}$$

$$= \{d, g, h, \epsilon\}$$

$$\text{FIRST}(B) = \text{FIRST}(g) \cup \text{FIRST}(\epsilon)$$

$$= \{g, \epsilon\}$$

$$\text{FIRST}(C) = \text{FIRST}(h) \cup \text{FIRST}(\epsilon)$$

$$= \{h, \epsilon\}$$

Computation of FOLLOW –

$$\text{FOLLOW}(S) = \{\$ \}$$

$$\text{FOLLOW}(A) = \{h, g, \$ \}$$

$$\text{FOLLOW}(B) = \{\$, a, h, g\}$$

$$\text{FOLLOW}(C) = \{\$, h, b, g\}$$

Prob.10. What is a recursive-descent parsing ? Define FIRST and FOLLOW functions. Consider the grammar –

$$S \rightarrow ACB|CbB|Ba$$

$$A \rightarrow da|BC$$

$$B \rightarrow g|\epsilon$$

$$C \rightarrow h|\epsilon$$

Calculate first and follow.

Sol. Refer to Q.10, Q.15 and Prob.9.

(R.G.P.V., May 2011)

Prob.11. For the following grammar find FIRST and FOLLOW sets for each of nonterminals –

$$S \rightarrow aAB | bA | \epsilon$$

$$A \rightarrow aAb | \epsilon$$

$$B \rightarrow bB | \epsilon.$$

(R.G.P.V., Dec. 2005, 2010)

Or

Show that the following grammar is LL(1) by constructing its parse table –

$$S \rightarrow aAB|bA|\epsilon$$

$$A \rightarrow aAb|\epsilon$$

$$B \rightarrow bB|\epsilon$$

(R.G.P.V., June 2009)

Sol. Computation of FIRST –

$$\text{FIRST}(S) = \text{FIRST}(aAB) \cup \text{FIRST}(bA) \cup \text{FIRST}(\epsilon)$$

$$= \{a\} \cup \{b\} \cup \{\epsilon\}$$

$$= \{a, b, \epsilon\}$$

$$\text{FIRST}(A) = \text{FIRST}\{aAB\} \cup \text{FIRST}(\epsilon)$$

$$= \{a\} \cup \{\epsilon\}$$

$$= \{a, \epsilon\}$$

$$\text{FIRST}(B) = \text{FIRST}(bB) \cup \text{FIRST}(\epsilon)$$

$$= \{b\} \cup \{\epsilon\}$$

$$= \{b, \epsilon\}$$

Computation of FOLLOW

$$\text{FOLLOW}(S) = \{\$ \}$$

$$\text{FOLLOW}(A) = \text{FIRST}(B) - \{\epsilon\} \cup \text{FOLLOW}(S) \cup \text{FIRST}(b)$$

$$\text{FOLLOW}(A) = \{b, \epsilon\} - \{\epsilon\} \cup \{\$ \} \cup \{b\}$$

$$\text{FOLLOW}(A) = \{b, \$ \}$$

$$\text{FOLLOW}(B) = \text{FOLLOW}(S) = \{\$ \}$$

LL(1) parsing table for the given grammar is shown in table 2.6.

Table 2.6 Parsing Table

	<i>a</i>	<i>b</i>	<i>\$</i>
S	$S \rightarrow aAB$	$S \rightarrow bA$	$S \rightarrow \epsilon$
A	$A \rightarrow aAb$	$A \rightarrow \epsilon$	$A \rightarrow \epsilon$
B		$B \rightarrow bB$	$B \rightarrow \epsilon$

Parsing table 2.6 contain no multiply-defined entries. So it is LL(1).

Prob.12. Consider the grammar –

$S \rightarrow aBDh$
 $B \rightarrow cC$
 $C \rightarrow bC|\epsilon$
 $D \rightarrow EF$
 $E \rightarrow g|\epsilon$
 $F \rightarrow f|\epsilon$

Construct the predictive parsing table.

(R.G.P.V., June 2011)

Sol. Computation of FIRST and FOLLOW –

$FIRST(S) = FIRST(aBDh) = \{a\}$

$FIRST(B) = FIRST(cC) = \{c\}$

$FIRST(C) = FIRST(bC) \cup FIRST(\epsilon) = \{b\} \cup \{\epsilon\} = \{b, \epsilon\}$

$FIRST(D) = FIRST(EF) = FIRST(E) - \{\epsilon\} \cup FIRST(F)$

$FIRST(E) = FIRST(g) \cup FIRST(\epsilon) = \{g, \epsilon\}$

$FIRST(F) = FIRST(f) \cup FIRST(\epsilon) = \{f, \epsilon\}$

Therefore by substituting in (i), we get

$FIRST(D) = \{g, \epsilon\} - \{\epsilon\} \cup \{f, \epsilon\} = \{g, f, \epsilon\}$

$FOLLOW(S) = \{\$ \}$

$FOLLOW(B) = FIRST(Dh) = FIRST(D) - \{\epsilon\} \cup FIRST(h)$
 $= \{g, f, \epsilon\} - \{\epsilon\} \cup \{h\} = \{g, f, h\}$

$FOLLOW(D) = FIRST(h) = \{h\}$

$FOLLOW(C) = FOLLOW(B) = \{g, f, h\}$

$FOLLOW(E) = FIRST(F) - \{\epsilon\} \cup FOLLOW(D)$

$= \{f, \epsilon\} - \{\epsilon\} \cup \{h\} = \{f, h\}$

$FOLLOW(F) = FOLLOW(D) = \{h\}$

Therefore, the parsing table is as shown in table 2.7.

Table 2.7 Parsing Table

	a	b	c	g	f	h	\$
S	$S \rightarrow aBDh$						
B			$B \rightarrow cC$				
C		$C \rightarrow bC$		$C \rightarrow \epsilon$	$C \rightarrow \epsilon$	$C \rightarrow \epsilon$	
D				$D \rightarrow EF$	$D \rightarrow EF$	$D \rightarrow EF$	
E				$E \rightarrow g$	$E \rightarrow \epsilon$	$E \rightarrow \epsilon$	
F					$F \rightarrow f$	$F \rightarrow \epsilon$	

Prob.13. Design LL(1) parsing table for the following grammar –

$A \rightarrow AcB|cC|C$

$B \rightarrow bB|id$

$C \rightarrow CaB|BbB|B$

(R.G.P.V., Dec. 2011)

Sol. As we see that the given grammar is left recursive. So, we first eliminate all left recursion from the grammar. Then, the resultant grammar is given as –

$A \rightarrow cCA'|CA'$

$A' \rightarrow cBA'|\epsilon$

$B \rightarrow bB|id$

$C \rightarrow BbBC'|BC'$

$C' \rightarrow aBC'|\epsilon$

Computation of FIRST and FOLLOW –

$FIRST(A) = FIRST(cCA') \cup FIRST(CA')$

$= \{c\} \cup FIRST(BbBC') \cup FIRST(BC')$

$= \{c\} \cup \{b, id\} \cup \{b, id\} = \{c, b, id\}$

$FIRST(A') = FIRST(cBA') \cup FIRST(\epsilon)$

$= \{c\} \cup \{\epsilon\} = \{c, \epsilon\}$

$FIRST(B) = FIRST(bB) \cup FIRST(id)$

$= \{b\} \cup \{id\} = \{b, id\}$

$FIRST(C) = FIRST(BbBC') \cup FIRST(BC')$

$= \{b, id\} \cup \{b, id\} = \{b, id\}$

$FIRST(C') = FIRST(aBC') \cup FIRST(\epsilon)$

$= \{a\} \cup \{\epsilon\} = \{a, \epsilon\}$

$FOLLOW(A) = \{\$ \}$

$FOLLOW(A') = FOLLOW(A) = \{\$ \}$

$FOLLOW(B) = \{c, \$, b, a\}$

$FOLLOW(C) = FIRST(A') - \{\epsilon\} \cup FOLLOW(A) = \{c, \epsilon\} - \{\epsilon\}$
 $\cup \{\$ \} = \{c, \$ \}$

$FOLLOW(C') = FOLLOW(C) = \{c, \$ \}$

LL(1) parsing table for the given grammar is shown in table 2.8.

Table 2.8 Parsing Table

	a	b	c	id	\$
A		$A \rightarrow CA'$	$A \rightarrow cCA'$	$A \rightarrow CA'$	
A'			$A' \rightarrow cBA'$		$A' \rightarrow \epsilon$
B		$B \rightarrow bB$		$B \rightarrow id$	
C		$C \rightarrow BbBC'$		$C \rightarrow BbBC'$	
C'	$C' \rightarrow aBC'$		$C' \rightarrow \epsilon$	$C \rightarrow BC'$	$C' \rightarrow \epsilon$

Prob.14. Test whether the grammar is LL(1) or not and construct predictive parsing table for it –

$S \rightarrow AaAb | BbBa$

$A \rightarrow \epsilon$

$B \rightarrow \epsilon$

(R.G.P.V., Dec. 2006)

Or

Show that the given grammar is LL(1) -

$$S \rightarrow AaAb|BbBa$$

$$A \rightarrow \epsilon$$

$$B \rightarrow \epsilon$$

(R.G.P.V., June 2012)

Or

Construct top-down parser for the grammar -

$$S \rightarrow AaAb|BbBa$$

$$A \rightarrow \epsilon$$

$$B \rightarrow \epsilon$$

(R.G.P.V., June 2012)

Sol. To construct a parsing table, the FIRST and FOLLOW sets are computed, as shown below -

$$\text{FIRST}(S) = \text{FIRST}(AaAb) \cup \text{FIRST}(BbBa)$$

$$\text{FIRST}(S) = \{a\} \cup \{b\} = \{a, b\}$$

$$\text{FIRST}(A) = \{\epsilon\}$$

$$\text{FIRST}(B) = \{\epsilon\}$$

$$\text{FOLLOW}(S) = \{\$ \}$$

(i) Using $S \rightarrow AaAb$, we get

$$\text{FOLLOW}(A) = \text{FIRST}(aAb) = \{a\}, \text{ and}$$

$$\text{FOLLOW}(A) = \text{FIRST}(b) = \{b\}, \text{ Therefore,}$$

$$\text{FOLLOW}(A) = \{a, b\}$$

(ii) Using $S \rightarrow BbBa$, we get

$$\text{FOLLOW}(B) = \text{FIRST}(bBa) = \{b\}, \text{ and}$$

$$\text{FOLLOW}(B) = \text{FIRST}(a) = \{a\}, \text{ Therefore,}$$

$$\text{FOLLOW}(B) = \{a, b\}$$

The grammar is LL(1), because the predictive parsing table contains no multiple defined entries.

Prob.15. Check whether the given grammar is LL(1) or not.

$$S \rightarrow iEtSS'/a$$

$$S' \rightarrow eS/E$$

$$E \rightarrow b$$

(R.G.P.V., Dec. 2011)

Sol. Computation of FIRST -

$$\text{FIRST}(S) = \text{FIRST}(iEtSS') \cup \text{FIRST}(a)$$

$$= \{i\} \cup \{a\}$$

$$= \{i, a\}$$

$$\text{FIRST}(S') = \text{FIRST}(eS) \cup \text{FIRST}(E)$$

$$= \{e\} \cup \{b\}$$

$$= \{e, b\}$$

$$\text{FIRST}(E) = \text{FIRST}(b)$$

$$= \{b\}$$

Computation of FOLLOW

$$\text{FOLLOW}(S) = \{e, b\}$$

$$\text{FOLLOW}(S') = \{e, b\}$$

$$\text{FOLLOW}(E) = \{t, e, b\}$$

LL(1) parsing table for the given grammar is shown in table 2.10.

Table 2.10

	i	t	a	e	b	\$
S	$S \rightarrow iEtSS'$		$S \rightarrow a$			
S'				$S' \rightarrow eS$	$S' \rightarrow E$	
E					$E \rightarrow b$	

The grammar is LL(1) because the predictive parsing table contains no multiple defined entries.

Prob.16. Consider the following grammar -

$$S \rightarrow IAB|\epsilon$$

$$A \rightarrow IAC|0C$$

$$B \rightarrow 0S$$

$$C \rightarrow I$$

and test whether the grammar is LL(1) or not. (R.G.P.V., June 2012)

Sol. Computation of FIRST and FOLLOW -

$$\text{FIRST}(S) = \text{FIRST}(IAB) \cup \text{FIRST}(\epsilon)$$

$$= \{I\} \cup \{\epsilon\}$$

$$= \{I, \epsilon\}$$

$$\text{FIRST}(A) = \text{FIRST}(IAC) \cup \text{FIRST}(0C)$$

$$= \{I\} \cup \{0\}$$

$$= \{I, 0\}$$

$$\text{FIRST}(B) = \text{FIRST}(0S)$$

$$= \{0\}$$

$$\text{FIRST}(C) = \text{FIRST}(I)$$

$$= \{I\}$$

$$\text{FOLLOW}(S) = \{\$ \}$$

$$\text{FOLLOW}(A) = \text{FIRST}(B) \cup \text{FIRST}(C)$$

$$= \{0\} \cup \{I\}$$

$$= \{0, I\}$$

$$\text{FOLLOW}(B) = \text{FOLLOW}(S)$$

$$= \{\$ \}$$

$$\text{FOLLOW}(C) = \text{FOLLOW}(A)$$

$$= \{0, I\}$$

LL(1) parsing table for the given grammar is shown in table 2.11.

Table 2.11 Parsing Table

	<i>I</i>	<i>0</i>	<i>\$</i>
S	$S \rightarrow 1AB$		$S \rightarrow \epsilon$
A	$A \rightarrow 1AC$	$A \rightarrow 0C$	
B		$B \rightarrow 0S$	
C	$C \rightarrow 1$		

Parsing table 2.11 contains no multiple-defined entries. So it is LL(1).

OPERATOR PRECEDENCE PARSING

Q.25. What do you understand by operator precedence parsing?
(R.G.P.V., June 2005, 2006)

Or

Write short note on operator precedence parsing. (R.G.P.V., Dec. 2006)

Or

Explain operator precedence parsing method with example.
(R.G.P.V., Dec. 2006)

Ans. Operator precedence parser works on a grammar called operator grammar. These grammars have the property that –

- No production right side is ϵ .
- No production right side has two adjacent non-terminals.

For example, the following grammar for expressions

$$E \rightarrow EAE \mid (E) \mid E \mid id$$

$$A \rightarrow + \mid - \mid * \mid / \mid \uparrow$$

is not an operator grammar, because the right side EAE has two consecutive nonterminals. However, if we substitute for A each of its alternatives, we obtain the following operator grammar –

$$E \rightarrow E + E \mid E - E \mid E * E \mid E / E \mid E \uparrow E \mid (E) \mid E \mid id$$

The operator precedence technique was first described as a manipulation on tokens without any reference to an underlying grammar. In fact, once we finish building an operator precedence parser from a grammar, we can effectively ignore the grammar, using the nonterminals on the stack only as placeholders for the nodes of a parse tree being constructed in the bottom-up fashion. Due to its simplicity, numerous compilers using operator precedence parsing techniques for expressions have been successfully built. Often the parsers use recursive descent for statements and higher-level constructs.

Operator precedence parsers have even been built for all languages. SNOBOL, being virtually “all operators” is an example of a language for which operator precedence works well.

In operator precedence parsing, we define three disjoint precedence relations, $<$, $=$, and $>$, between certain pairs of terminals. These precedence relations guide the selection of handles and have the following meanings –

Relation	Meaning
$a < b$	a “yields precedence to” b
$a = b$	a “has the same precedence as” b
$a > b$	a “takes precedence over” b

There are following two common ways of determining what precedence relation should hold between a pair of terminals –

(i) This method is intuitive and is based on the traditional notions of associativity and precedence of operators. For example, if $*$ is to have higher precedence than $+$, we make $+ < *$ and $* > +$. This approach will be seen to resolve the ambiguities of grammar and to enable us to write an operator precedence parser for it.

(ii) In this method, we construct an unambiguous grammar for the language, a grammar which reflects the correct associativity and precedence in its parse tree.

Q.26. Write a short note on operator precedence parsing and function.
(R.G.P.V., Nov. 2018)

Ans. Operator Precedence Parsing – Refer to Q.25.

Operator Precedence Function – Compilers using operator precedence parsers need not to store the table of precedence relations. Mostly the table can be encoded by two precedence functions f and g that map terminal symbols to integers. We attempt to select f and g for symbols a and b , so that –

- $f(a) < g(b)$ whenever $a < b$,
- $f(a) = g(b)$ whenever $a = b$,
- $f(a) > g(b)$ whenever $a > b$.

Thus, the precedence relation between a and b can be determined by a numerical comparison between $f(a)$ and $g(b)$. However, it is to be noted that error entries in the precedence matrix are obscured, since one of (i), (ii) or (iii) holds no matter what $f(a)$ and $g(b)$ are. Generally, the loss of error detection capability is not considered serious enough to prevent the using of precedence functions where possible; errors can still be caught when a reduction is called for and no handle can be found.

Every table of precedence relations has not functions to encode it, but in practical cases the functions usually exist.

Q.27. How does an operator precedence parser work? Use a preconstructed operator precedence table to guide the parsing of an input 'a + b - 20' using operator precedence parser. (R.G.P.V., Dec. 2013)

Ans. Refer to Q.25.

Consider an operator grammar recognising expressions using the following seven productions –

$$E \rightarrow E + E \mid E - E \mid E * E \mid E / E \mid E \uparrow E \mid (E) \mid -E \mid id$$

The id can be either a variable or a number. The operator precedence relation table for the above grammar is shown in table 2.12.

Table 2.12 Operator Precedence Relation Table

	+	-	*	/	\uparrow	()	id	\$
+	>	>	<	<	<	<	>	<	>
-	>	>	<	<	<	<	>	<	>
*	>	>	>	>	<	<	>	<	>
/	>	>	>	>	<	<	>	<	>
\uparrow	>	>	>	>	<	<	>	<	>
(<	<	<	<	<	<	=	<	
)	>	>	>	>	>		>		>
id	>	>	>	>	>		>		>
\$	<	<	<	<	<	<		<	

Using the above grammar and corresponding precedence table, the operator precedence parser algorithm for an input 'a + b - 20' works as follows –

Stack	Input
\$	a + b - 20 \$
\$ id	+ b - 20 \$
\$ E	+ b - 20 \$
\$ E +	b - 20 \$
\$ E + id	- 20 \$
\$ E + E	- 20 \$
\$ E	- 20 \$
\$ E -	20 \$
\$ E - id	\$
\$ E - E	\$
\$ E	\$

Now, the top most terminal on the stack is \$ and also the next input symbol is \$, which signals the completion of a successful parse.

Q.28. What are the advantages of operator precedence parsing?

Ans. Following are the advantages of operator precedence parsing –

- It is simple and easy to implement parsing technique.
- The operator precedence parser is constructed by hand after understanding the grammar. It is simpler to debug.

Q.29. What are the limitations of operator precedence parsing?

(R.G.P.V., Dec. 2002, June 2016)

Ans. The limitations of operator precedence parsing are as follows –

- In operator precedence parsing it is hard to handle tokens like the minus sign, which has two different precedence (depending on whether it is unary or binary).
- Worse, since the relationship between a grammar for the language being parsed and the operator precedence parser itself is tenuous, one cannot always be sure the parser accepts exactly the desired language.
- Only a small class of grammars can be parsed using operator precedence techniques.

Q.30. Describe operator precedence parsing algorithm.

(R.G.P.V., Dec. 2005, June 2007)

Ans. Operator precedence parsing algorithm is as follows –

Input – An input string w and a table of precedence relations.

Output – If w is well formed, a skeletal parse tree, with a placeholder nonterminal E labeling all interior nodes; otherwise, an error indication.

Method – Initially, the stack contains \$ and the input buffer contains the string w\$. To parse, we execute the program of fig. 2.9.

```

(1) set ip to point to the first symbol of w$.
(2) repeat forever
(3)   if $ is on top of the stack and ip points to $ then
(4)     return
(5)   else begin
(6)     let a be the topmost terminal symbol on the stack
(7)     and let b be the symbol pointed to by ip;
(8)     if a < b or a = b then begin
(9)       push b onto the stack;
(10)      advance ip to the next input symbol;
(11)    end;
(12)  else if a > b then /* reduce */
(13)    repeat
(14)      pop the stack
(15)    until the top stack terminal is related by <
(16)    to the terminal most recently popped
(17)  else error( )
(18) end
  
```

Fig. 2.9 Operator Precedence Parsing Algorithm

Q.31. Write the algorithm for calculation of operator precedence relations. (R.G.P.V., Dec. 2008)

Ans. Algorithm for computation of operator-precedence relations is as follows.

Input – An operator grammar G

Output – The relations $<$, \doteq and $>$ for G

Method –

- (i) Compute LEADING(A) and TRAILING(A) for each nonterminal A
- (ii) Execute the program of fig. 2.10 examining each position of the right side of each production.

```

for each production  $A \rightarrow X_1 X_2 \dots X_n$  do
  for  $i := 1$  to  $n-1$  do
    begin
      if  $X_i$  and  $X_{i+1}$  are both terminals then set  $X_i \doteq X_{i+1}$ ;
      if  $i \leq n-2$  and  $X_i$  and  $X_{i+2}$  are terminals
        and  $X_{i+1}$  is a nonterminal then
        set  $X_i \doteq X_{i+2}$ ;
      if  $X_i$  is a terminal and  $X_{i+1}$  is a nonterminal then
        for all  $a$  in LEADING( $X_{i+1}$ ) do set  $X_i < a$ ;
      if  $X_i$  is a nonterminal and  $X_{i+1}$  is a terminal then
        for all  $a$  in TRAILING( $X_i$ ) do set  $a > X_{i+1}$ 
    end
  end

```

Fig. 2.10 Calculation of Operator-precedence Relations

- (iii) Set $\$ < a$ for all a in LEADING(S) and set $b > \$$ for all b in TRAILING(S), where S is the start symbol of G .

Q.32. What is operator precedence grammar? (R.G.P.V., June 2008)

Ans. An operator precedence grammar is defined as an ϵ free operator grammar in which the precedence relation $<$, \doteq and $>$ constructed are disjoint. That is for a pair of terminal a and b never more than one of the relations $a > b$, $a < b$ and $a \doteq b$ is true. The precedence relation among terminals is deduced and the precedence table is constructed. The procedure applied in doing so is as stated below. For each two terminal symbols a and b , we say

- (i) $a \doteq b$ if there is a right side of production of the form $\alpha a \beta b$ where β is either ϵ or a single nonterminal i.e., $a \doteq b$ if a appears immediately to the left of b in a right side, or if they appear separated by one nonterminal
- (ii) $a < b$ if some nonterminal A there is a right side of the form $\alpha A \beta$ and $A \xrightarrow{\doteq} \gamma b \delta$, where γ is either ϵ or a single nonterminal. So, $a < b$ if a nonterminal A appears immediately to the right of ' a ' and derives a string which b is the first terminal symbol.
- (iii) $a > b$ if for some non-terminal A there is a right side of the form $\alpha A \beta$ and $A \xrightarrow{\doteq} \gamma a \delta$, where δ is either ϵ or a single nonterminal so $a > b$ if ' a ' nonterminal appearing immediately to the left of ' b ' derives ' a ' string whose last terminal is a .

The above three can be easily applied by using two functions namely LEADING(A) and TRAILING(A), which can be defined as below

LEADING(A) = $\{a | A \xrightarrow{\doteq} \gamma a \delta, \text{ where } \gamma \text{ is } \epsilon \text{ or a single nonterminal}\}$

TRAILING(A) = $\{a | A \xrightarrow{\doteq} \gamma a \delta, \text{ where } \delta \text{ is } \epsilon \text{ or a single nonterminal}\}$.

Leading(A) is the set which consist of all terminals that can be first terminal in a string derived from 'A'. Trailing(A) is the set which consist of all terminals that can be last terminal in a string derived from 'A'. For example consider a operator grammar.

Table 2.13 First and Last Terminals

	Nonterminal	First Terminal	Last Terminal
$E \rightarrow E + T T$	E	*, +, (, id	*, +,), id
$T \rightarrow T * F F$	T	*, (, id	*,), id
$F \rightarrow (E) id$	F	(, id), id

- (i) To compute the \doteq relation we look for right side in production where two terminals are adjacent or with a single nonterminal in between. For example (\doteq).

- (ii) To compute $<$ we see the R.H.S. for terminals ' a ' which are followed by nonterminals ' A '. For every such pair ' a ' is related by $<$ to any terminal which comes in the leading list of ' A '. For example, $+$ and T , $*$ and F and, $($ and E .

- (iii) The $>$ is computed in the same manner for every pair of non-terminal followed by terminal ' b ' by making use of trailing list.

Q.33. Describe the error reporting and recovery schemes in operator precedence parsing. (R.G.P.V., Dec. 2012)

Ans. In operator precedence parsing process, there are two points at which an operator precedence parser can detect errors. First, when no precedence relation exists between the terminal on top of the stack and current input symbol. In these situations, the source of error can be determined by virtue of the position in the operator precedence table. For instance, the entry $O[id][id]$ is referenced when the input has two consecutive ids without using an operator between them. In such cases, we can issue a diagnostic message 'Missing Operand'. In similar manner, some other messages such as 'Missing Operator', 'Missing Left Parenthesis' and 'Missing Right Parenthesis' can also be flashed on the position in the relation table. In all of these situations where we can flash the error messages, we can also proceed with syntax analysis of the next line without stopping at the error by filling in the missing element on the stack. For example, in case of 'Missing Right Parenthesis' message, we can insert a right parenthesis in the stack and allow the syntax analysis to proceed.

Second, when a handle has been found, but there is no production with this handle as a right side. In these situations, we could flash a message depending on the resemblance to a particular production. For example, suppose a handle is 'B*', the production that it resembles is ' $A \rightarrow B * C$ '. Here, the missing element is another id, which would have been reduced to A. In this case, a diagnostic message is issued showing a 'Missing Operand' message. Generally, for the grammar, we can potentially check if there are non terminals on either end of other operators, if they do not exist, then we can issue a diagnostic message 'Missing Operand'. For error recovery, we can insert the id on the stack and then proceed. We can also deduce that a non-terminal needs to be present between the parentheses or else we can flash a diagnostic message indicating that the expression is missing between the parentheses.

NUMERICAL PROBLEMS

Prob.17. Find LEADING and TRAILING of the following grammar.

$$S \rightarrow aAB|bA| \epsilon$$

$$A \rightarrow aAb| \epsilon$$

$$B \rightarrow bB|C$$

(R.G.P.V., June 2008)

Sol. Given grammar is not an operator grammar, so we can transform the given grammar into an equivalent grammar.

Table 2.14

$$S \rightarrow aaAbB|aabB|bA|aB|b$$

$$A \rightarrow aAb|ab$$

$$B \rightarrow bB|C$$

Nonterminal	LEADING	TRAILING
S	a, b	a, b
A	a	b
B	b	b

The LEADING and TRAILING for the above grammar are computed by-

$$\text{LEADING}(X) = \{a \mid X \xRightarrow{+} \gamma a \delta, \text{ where } \gamma \text{ is } \epsilon \text{ or a single nonterminal}\}$$

$$\text{TRAILING}(X) = \{a \mid X \xRightarrow{+} \gamma a \delta, \text{ where } \delta \text{ is } \epsilon \text{ or a single nonterminal}\}$$

Prob.18. Find LEADING and TRAILING of the following grammar.

$$S \rightarrow aAB|bA| \epsilon$$

$$A \rightarrow aAb| \epsilon$$

$$B \rightarrow bB|c$$

(R.G.P.V., June 2012)

Sol. Given grammar is not an operator grammar, so we can transform the given grammar into an equivalent operator grammar. In operator grammar, no production of right side is ϵ and no production of right side contains two adjacent non-terminals. Elimination of adjacent non-terminals is performed by putting all the alternatives of the non-terminal.

$$S \rightarrow aaAbB|aabB|aAbB|aAc|bA|aB|b$$

$$A \rightarrow aAb|ab$$

$$B \rightarrow bB|c$$

The LEADING and TRAILING for the above grammar are shown in table 2.15.

Table 2.15

Nonterminal	LEADING	TRAILING
S	a, b	b, c, a
A	a	b
B	b, c	b, c

Prob.19. Construct operator precedence parser and then parse the following string -

$$S \rightarrow (L) | a \quad \text{string } "(a,(a,a))"$$

$$L \rightarrow L, S | S$$

(R.G.P.V., Dec. 2008)

Sol. The given grammar

$$S \rightarrow (L) | a$$

$$L \rightarrow L, S | S$$

is operator grammar. LEADING(A) and TRAILING(A) for each nonterminal A calculated are shown in table 2.16.

Table 2.16

Nonterminal	LEADING	TRAILING
S	(, a), a
L	,, (, a	,,), a

Table 2.17 Precedence Relations

	()	a	,	\$
(<	=	<	<	<
)	>	>	>	>	>
a	>	>	>	>	>
,	<	>	<	>	<
\$	<	<	<	<	<

Operator precedence relations for the given grammar are shown in table 2.17. The sequence of shift-reduce steps for the given input string (a, (a, a)) is as follows -

Stack	Input
\$	< (a, (a, a))\$
\$ < (a, (a, a) \$
\$ < (< a	, (a, a) \$
\$ < (S	, (a, a) \$
\$ < (< S,	(a, a) \$
\$ < (< S, < (a, a) \$
\$ < (< S, < (< a	, a) \$
\$ < (< S, < (S	, a) \$
\$ < (< S, < (< S,	a) \$
\$ < (< S, < (< S, < a) \$
\$ < (< S, < (< S, S) \$

\$ < (< S, < (< S, S >)) \$	=)) \$
\$ < (< S, < (< S, S >))	=)) \$
\$ < (< S, < (< S, S >) >)	=)) \$
\$ < (< S, < (< S, S >) > >)	=)) \$
\$ < (< S, < (< S, S >) > >)	=)) \$
\$ < (< S, < (< S, S >) > >)	=)) \$
\$ < (< S, < (< S, S >) > >)	=)) \$

Prob.20. Compute the operator precedence relations for the grammar.

$$S \rightarrow a \mid \wedge \mid (T)$$

$$T \rightarrow T, S \mid S$$

Is it an operator precedence grammar?

(R.G.P.V., Dec. 2009)

Or

Construct operator precedence parser.

$$S \rightarrow a \mid \wedge \mid (T)$$

$$T \rightarrow T, S \mid S$$

(R.G.P.V., June 2014)

Sol. The given grammar is not an operator grammar, so we can transform the given grammar into an equivalent grammar that is both operator grammar and unambiguous grammar

$$S \rightarrow (T) \mid a$$

$$T \rightarrow T, S \mid S$$

Operator Precedence Relations – Refer to Prob.19, table 2.17.

Yes, it is an operator precedence grammar.

LR PARSERS (SLR, LALR, LR), PARSER GENERATION

Q.34. What do you understand by LR(k) grammar?

(R.G.P.V., Dec. 2005, 2007)

Ans. A grammar for which we can construct a parsing table in which every entry is uniquely defined is said to be an LR grammar. For an LR grammar, it is sufficient that a left-to-right parser be able to recognize handles when they appear on top of the stack.

An LR parser does not have to scan the entire stack to know when the handle appears on top. Rather, the state symbol on top of the stack contains all the information it needs. It is a remarkable fact that if it is possible to recognize a handle knowing only what is in the stack, then a finite automaton can, by reading the stack from bottom to top, determine what handle, if any, is on top of the stack. The goto function of an LR parsing table is essentially such a finite automaton. The automaton need not, however, read the stack on every move. The state symbol stored on top of the stack is the state the handle

recognizing finite automaton would be in if it had read the grammar symbols of the stack from bottom to top. Thus, the LR parser can determine from the state on top of the stack everything that it need to know about what is in the stack.

Another source of information that an LR parser can use to help make its shift-reduce decisions is the next k input symbols. The cases $k = 0$ or $k = 1$ are of practical interest, and we shall only consider LR parser with $k \leq 1$. A grammar that can be parsed by an LR parser examining up to k input symbols on each move is called an LR(k) grammar. The L is for left-to-right scanning of the input, the R for constructing a rightmost derivation in reverse and the k for the number of input symbols of lookahead that are used in making parsing decisions. When (k) is omitted, k is assumed to be 1.

Q.35. Explain the various steps for constructing SLR parsing tables.

Or

Describe the SLR(1) method of constructing the LR parsing table from a given grammar. Illustrate with an example. (R.G.P.V., Dec. 2015)

Ans. SLR also known as "simple LR", is the easiest method to implement. A grammar for which an SLR parser can be constructed is said to be an SLR grammar. An LR(0) item of a grammar G is a production of G with a dot at some position of the right side. Thus, production $A \rightarrow XYZ$ yields the following items

$$A \rightarrow \cdot XYZ$$

$$A \rightarrow X \cdot YZ$$

$$A \rightarrow XY \cdot Z$$

$$A \rightarrow XYZ \cdot$$

The production $A \rightarrow \epsilon$ generates $A \rightarrow \cdot$. An item can be represented by a pair of integers, the first giving the number of the production and second the position of the dot. Intuitively, an item indicates how much of a product we have seen at a given point in the parsing process.

The SLR method is used to construct from the grammar a deterministic finite automaton to recognize viable prefixes. Items are grouped into sets which give rise to the states of the SLR parsers. Canonical LR(0) collection is used for constructing SLR parsers, for this we define augmented grammar and two functions, CLOSURE and GOTO.

(i) **Augmented Grammar** – If G is a grammar with start symbol S , then G' is the augmented grammar for G , the G with a new start symbol S' and production $S' \rightarrow S$. This indicates the completion of parsing and acceptance of the input. The acceptance occurs when parser is about to reduce by $S' \rightarrow S$.

(ii) **Closure Operation** – If I is a set of items for a grammar G , then CLOSURE(I) is constructed from I by two rules –

(a) Every item in I is added to CLOSURE(I)

(b) If $A \rightarrow \alpha \cdot B \beta$ is in CLOSURE(I) and $B \rightarrow \gamma$ is a production, then add $B \rightarrow \cdot \gamma$ to I . Apply this until no more new items can be added to

CLOSURE(I). For example –

$E' \rightarrow E$

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

then CLOSURE(I) contains the item,

$E' \rightarrow \cdot E$

$E \rightarrow \cdot E + T$

$E \rightarrow \cdot T$

$T \rightarrow \cdot T * F$

$T \rightarrow \cdot F$

$F \rightarrow \cdot (E)$

$F \rightarrow \cdot id$

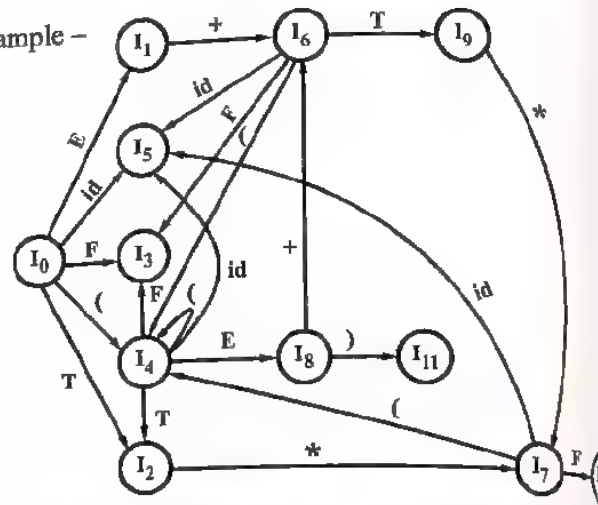


Fig. 2.11 Deterministic Finite Automaton D

(iii) **GOTO Operation** – $GOTO(I, x)$ is defined to be the closure of the set of all items $[A \rightarrow \alpha X \cdot \beta]$ such that $[A \rightarrow \alpha \cdot X \beta]$ is in I , where I is a set of items and X is a grammar symbol. If I is the set of valid items for some viable prefix γ , then $GOTO(I, X)$ is the set of items that are valid for the viable prefix γX . The GOTO function can be applied on the above grammar in the following way.

Canonical LR(0) collection for grammar are –

$I_0: E' \rightarrow \cdot E$

$E \rightarrow \cdot E + T$

$E \rightarrow \cdot T$

$T \rightarrow \cdot T * F$

$T \rightarrow \cdot F$

$F \rightarrow \cdot (E)$

$F \rightarrow \cdot id$

$I_1: E' \rightarrow E \cdot$

$E \rightarrow E \cdot + T$

$I_2: E \rightarrow T \cdot$

$T \rightarrow T \cdot * F$

$I_3: T \rightarrow F \cdot$

$I_4: F \rightarrow (\cdot E)$

$E \rightarrow \cdot E + T$

$E \rightarrow \cdot T$

$T \rightarrow \cdot T * F$

$T \rightarrow \cdot F$

$F \rightarrow \cdot (E)$

$F \rightarrow \cdot id$

$I_5: F \rightarrow id \cdot$

$I_6: E \rightarrow E + \cdot T$

$T \rightarrow \cdot T * F$

$T \rightarrow \cdot F$

$F \rightarrow \cdot (E)$

$F \rightarrow \cdot id$

$I_7: T \rightarrow T * \cdot F$

$F \rightarrow \cdot (E)$

$F \rightarrow \cdot id$

$I_8: F \rightarrow (E \cdot)$

$E \rightarrow E \cdot + T$

$I_9: E \rightarrow E + T \cdot$

$T \rightarrow T \cdot * F$

$I_{10}: T \rightarrow T * F \cdot$

$I_{11}: F \rightarrow (E) \cdot$

G to produce G' and from G' we construct C , the canonical collection of sets of items for G' . Action and GOTO functions can be constructed from C using the following SLR parsing table construction technique.

Algorithm – Construction of an SLR parsing table.

Input. An augmented grammar G' .

Output. An SLR parsing table functions action and goto for G' .

Steps (i) $C = \{I_0, I_1, \dots, I_n\}$. The states of the parser are $0, 1, \dots, n$, state j being constructed from I_i .

(ii) The parsing actions for state i are determined as follows –

(a) If $[A \rightarrow \alpha \cdot a\beta]$ is in I_i and $GOTO(I_i, a) = I_j$, then set ACTION[i, a] to “shift j ” where a must be a terminal.

(b) If $[A \rightarrow \alpha \cdot]$ is in I_i , then set action [i, a] to “reduce $A \rightarrow \alpha$ ” for all a in FOLLOW(A).

(c) If $[S' \rightarrow S \cdot]$ is in I_i , then set ACTION[i, \$] to “accept”.

If any conflicting actions are generated by the above rules, we say the grammar is not SLR(1).

The goto transitions for state i are constructed using the following rules –

(iii) If $GOTO(I_i, A) = I_j$, then $GOTO[i, A] = j$

(iv) All entries not defined by rules (ii) and (iii) are made “error”.

(v) The initial state of the parser is the one constructed from the set of items containing $[S' \rightarrow \cdot S]$.

Table 2.18 Parsing Table for Expressing Grammar

State	Action						Goto		
	<i>id</i>	<i>+</i>	<i>*</i>	<i>(</i>	<i>)</i>	<i>\$</i>	<i>E</i>	<i>T</i>	<i>F</i>
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

The parsing table consisting of the parsing action and goto functions is called the SLR(1) table for G . The SLR parsing table for the given example is shown in table 2.18. An LR parser using the SLR(1) table for G is called the SLR(1) parser for G and a grammar having an SLR(1) parsing table is said to be SLR(1).

SLR Parsing Tables – SLR parsing action and goto function can be constructed from the DFA that recognizes viable prefixes. Given a grammar

For constructing the SLR parsing table, the canonical collection of sets of LR(0) items.

The item $F \rightarrow \cdot (E)$ gives rise to the entry $\text{ACTION}[0, (] = \text{shift } 4$, and item $F \rightarrow \cdot \text{id}$ to the entry $\text{ACTION}[0, \text{id}] = \text{shift } 5$. Other items in I_0 yield no actions. Now consider I_1 -

$$E' \rightarrow E.$$

$$E \rightarrow E \cdot + T$$

The first item yields $\text{ACTION}[1, \$] = \text{accept}$, the second item yields $\text{ACTION}[1, +] = \text{shift } 6$. For I_2 -

$$E \rightarrow T \cdot$$

$$T \rightarrow T \cdot * F$$

Since, $\text{FOLLOW}(E) = \{\$, +,)\}$, the first item makes $\text{ACTION}[2, \$] = \text{accept}$, $\text{ACTION}[2, +] = \text{ACTION}[2,)] = \text{reduce } E \rightarrow T$. The second item makes $\text{ACTION}[2, *] = \text{shift } 7$.

Continuing in this fashion we obtain the parsing action and goto table. The numbers of productions in reduce actions are the same as the order in which they appear in the original grammar. That is, $E \rightarrow E + T$ is number 1, $E \rightarrow T$ is 2, and so on.

Q.36. What are the merits and demerits of LR-parser? Construct LR parsing table for the following grammar with necessary codes for action

$$E \rightarrow E + T \mid T$$

$$T \rightarrow T * F \mid F$$

$$F \rightarrow (E) \mid \text{id}$$

(R.G.P.V., Dec. 2015)

Ans. The merits of LR-parser are as follows -

- It is non recursive.
- It is the most general nonbacktracking technique known.
- An LR-parser can recognize virtually all programming language constructs written with context-free grammars.
- It can detect syntax errors as soon as it is possible in a left-to-right scan.
- It can be implemented in a very efficient manner.

The only disadvantage to LR parsers is that it requires hard work to manually create LR parsing tables.

Also, refer to Q.35.

Q.37. Discuss error recovery in LR parser.

Ans. An LR parser will detect an error when it consults the parsing action table and find a blank or error entry. Errors are never detected by consulting

the goto table. An LR parser will detect an error as soon as there is no valid continuation for the portion of the input so far scanned. A CLR parser will not make even a single reduction before announcing the error. SLR and LALR parsers may make several reductions before detecting an error, but they will never shift an erroneous input symbol onto the stack.

Q.38. How to construct an CLR parser? Explain.

Ans. Canonical LR parser is the most general technique for constructing an LR parsing table from a grammar. In SLR method, state i calls for reduction by $A \rightarrow \alpha$ if the set of items I_i contains item $[a \rightarrow \alpha \cdot]$ and a is in $\text{FOLLOW}(A)$. In some situations, however, when state i appears on top of the stack, the viable prefix $\beta\alpha$ on the stack is such that βA cannot be followed by a in a right-sentential form. Thus, the reduction by $A \rightarrow \alpha$ would be invalid on input a .

It is possible to carry more information in the state that will allow us to rule out some of these invalid reductions by $A \rightarrow \alpha$. By splitting states when necessary, we can arrange to have each state of an LR parser indicate exactly which input symbols can follow a handle α for which there is a possible reduction to A .

The extra information is incorporated into the state by redefining items to include a terminal symbol as a second component. The general form of an item becomes $[A \rightarrow \alpha \cdot \beta, a]$, where $A \rightarrow \alpha\beta$ is a production and a is a terminal or the right endmarker $\$$. We call such an object an LR(1) item. The 1 refers to the length of the second component, called the lookahead of the item. The lookahead has no effect in an item of the form $[A \rightarrow \alpha \cdot \beta, a]$, where β is not ϵ , but an item of the form $[A \rightarrow \alpha \cdot, a]$ calls for a reduction by $A \rightarrow \alpha$ only if the next input symbol is a . Thus, we are compelled to reduce by $A \rightarrow \alpha$ only on those input symbols for which $[A \rightarrow \alpha \cdot, a]$ is an LR(1) item in the state on top of the stack.

Formally, we say LR(1) item $[A \rightarrow \alpha \cdot \beta, a]$ is valid for a viable prefix γ if there is a derivation $S \xRightarrow{*}_{rm} \delta A w \xRightarrow{*}_{rm} \delta \alpha \beta w$, where

$$(i) \quad \gamma = \delta \alpha$$

$$(ii) \quad \text{Either } a \text{ is the first symbol of } w, \text{ or } w \text{ is } \epsilon \text{ and } a \text{ is } \$.$$

The method for constructing the collection of sets of valid LR(1) items is similar to the LR(0) items. The need to modify the two procedures CLOSURE and GOTO.

To appreciate the new definition of the closure operation, consider an item of the form $[A \rightarrow \alpha \cdot B, \beta a]$ in the set of items valid for some viable prefix γ . Then there is a rightmost derivation $S \xRightarrow{*}_{rm} \delta A \alpha x \xRightarrow{*}_{rm} \delta \alpha B \beta \alpha x$, where $\gamma = \delta \alpha$. Suppose $\beta \alpha x$ derives terminal string y . Then for each production of the form $B \rightarrow \eta$ for some η , we have derivation $S \xRightarrow{*}_{rm} \gamma B \beta y \xRightarrow{*}_{rm} \gamma \eta \beta y$. Thus,

$[B \rightarrow \cdot \eta, b]$ is valid for γ . We say that b can be any terminal in $\text{FIRST}(\beta a)$. We now give the LR(1) set of items construction.

Input – An augmented grammar G' .

Output – The sets of LR(1) items that are the set of items valid for one or more viable prefixes of G' .

Method – The procedures CLOSURE and GOTO and the main routine items for constructing the sets of items are shown in fig. 2.12.

```

function closure(I);
begin
  repeat
    for each item  $[A \rightarrow \alpha \cdot B \beta, a]$  in  $I$ ,
      each production  $B \rightarrow \gamma$  in  $G'$ ,
      and each terminal  $b$  in  $\text{FIRST}(\beta a)$ 
      such that  $[B \rightarrow \gamma, b]$  is not in  $I$  do
        add  $[B \rightarrow \gamma, b]$  to  $I$ ;
    until no more items can be added to  $I$ ;
  return I
end;

function goto(I, X);
begin
  let  $J$  be the set of items  $[A \rightarrow \alpha X \cdot \beta, a]$  such that
     $[A \rightarrow \alpha \cdot X \beta, a]$  is in  $I$ ;
  return closure(J)
end;

procedure items( $G'$ );
begin
   $C := \{\text{closure}(\{[S' \rightarrow \cdot S, \$])\}$ 
  repeat
    for each set of items  $I$  in  $C$  and each grammar symbol  $X$ 
      such that  $\text{goto}(I, X)$  is not empty and not in  $C$  do
        add  $\text{goto}(I, X)$  to  $C$ 
  until no more sets of items can be added to  $C$ 
end

```

Fig. 2.12 Sets of LR(1) Items Construction for Grammar G'

Consider the following augmented grammar –

$$\begin{aligned} S' &\rightarrow S \\ S &\rightarrow CC \\ C &\rightarrow cC \mid d \end{aligned}$$

The canonical collection of LR(1) items is as follows –

$$\begin{aligned} I_0 : S' &\rightarrow \cdot S, \$ & I_4 : C &\rightarrow d \cdot, c/d \\ S &\rightarrow \cdot CC, \$ & I_5 : S &\rightarrow CC \cdot, \$ \\ C &\rightarrow \cdot cC, c/d & I_6 : C &\rightarrow c \cdot C, \$ \\ C &\rightarrow \cdot d, c/d & C &\rightarrow \cdot cC, \$ \end{aligned}$$

$$\begin{aligned} I_1 : S' &\rightarrow S \cdot, \$ & C &\rightarrow \cdot d, \$ \\ I_2 : S &\rightarrow C \cdot C, \$ & I_7 : C &\rightarrow d \cdot, \$ \\ C &\rightarrow \cdot cC, \$ & I_8 : C &\rightarrow cC \cdot, c/d \\ C &\rightarrow \cdot d, \$ & I_9 : C &\rightarrow cC \cdot, \$ \\ I_3 : C &\rightarrow c \cdot C, c/d & & \\ C &\rightarrow \cdot cC, c/d & & \\ C &\rightarrow \cdot d, c/d & & \end{aligned}$$

Fig. 2.13 shows the ten sets of items with their goto's.

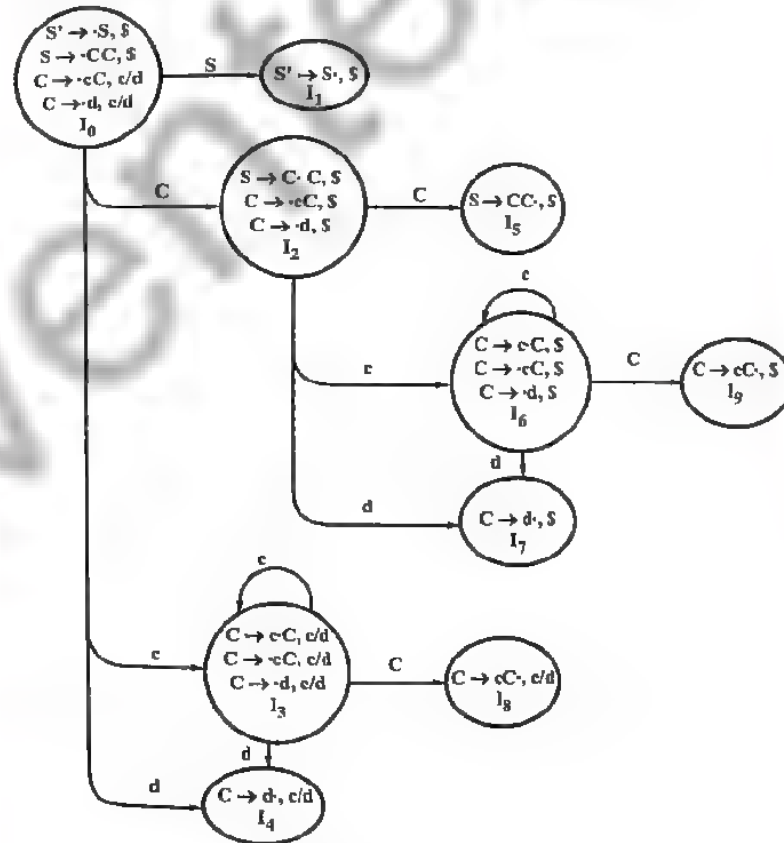


Fig. 2.13 The GOTO Graph

Q.39. Write the Algo/Procedure for construction of the canonical LR parsing table. (R.G.P.V., June 2004)

Ans. Algorithm – Construction of canonical LR parsing table.

Input – An augmented grammar G' .

Output – The canonical LR parsing table functions ACTION and GOTO for G' .

Method – (i) Construct $C = \{I_0, I_1, \dots, I_n\}$, the collection of sets of LR(1) items for G' .

(ii) State i of the parser is constructed from I_i . The parsing actions for state i are determined as follows –

(a) If $[A \rightarrow \alpha a \beta, b]$ is in I_i and $\text{GOTO}(I_i, a) = I_j$ then set $\text{ACTION}[i, a]$ to “Shift j ”.

(b) If $[A \rightarrow \alpha, a]$ is in I_i , then set $\text{ACTION}[i, a]$ to “reduce $A \rightarrow \alpha$ ”.

(c) If $[S' \rightarrow S, \$]$ is in I_i , then set $\text{ACTION}[i, \$]$ to “accept”.

(iii) The goto transitions for state i are determined as follows –

If $\text{GOTO}(I_i, A) = I_j$, then $\text{GOTO}[i, A] = j$.

(iv) All entries not defined by rules (ii) and (iii) are made “error”.

(v) The initial state of the parser is the one constructed from the set containing item $[S' \rightarrow \cdot S, \$]$.

Table 2.19 Canonical Parsing Table

The table formed is called canonical LR(1) parsing table. An LR parser using this table is called a canonical LR parser. If the parsing action function has no multiple-defined entries, then the given grammar is called an LR(1) grammar. The canonical parsing table for the grammar (i), given in Q.38 is shown in table 2.19.

STATE	ACTION			GOTO	
	c	d	\$	S	C
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

Q.40. Describe the construction of LALR parsing table.

Ans. The another parser construction method, LALR (Lookahead-LR) technique is often used because the tables obtained by it are considerably smaller than the canonical LR tables.

The SLR and LALR parsing tables for a grammar always have the same number of states, typically several hundred states. The LALR parsing table is derived from the canonical LR table by merging the states which have similar cores. The new state thus formed have a number where digit represents the actual states which are merged. Let us consider the grammar (i), from Q.38 whose sets of LR(1) items were shown in fig. 2.12. Take a pair of similar looking states, such as I_4 and I_7 . Each of these states has only items with first component $C \rightarrow d \cdot$. In I_4 , the lookaheads are c or d ; in I_7 , $\$$ is the only lookahead.

Let us now replace I_4 and I_7 by I_{47} , the union of I_4 and I_7 , consisting of the set of three items represented by $[C \rightarrow d \cdot, c/d/\$]$. The goto's on d to I_4 and I_7 from I_0, I_2, I_3 and I_6 now enter I_{47} . The action of state 47 is to reduce C on

any input. The revised parser behaves essentially like the original, although it might reduce d to C in circumstances where the original would declare error, for example, on input like ccd or $cdcdc$. The error will eventually be caught; in fact, it will be caught before any more input symbols are shifted. Similarly I_3 and I_6 form another pair, with core $\{C \rightarrow c \cdot C, C \rightarrow \cdot cC, C \rightarrow \cdot d\}$. There is one more pair, I_8 and I_9 , with core $\{C \rightarrow cC \cdot\}$.

Since the core of $\text{goto}(I, X)$ depends only on the core of I , the goto's of merged sets can themselves be merged. Thus, there is no problem revising the goto function as we merge sets of items. The action functions are modified to reflect the non-error actions of all sets of items in the merger.

Suppose we have an LR(1) grammar, that is, one whose sets of LR(1) items produce no parsing action conflicts. If we replace all states having the same core with their union, it is possible that the resulting union will have a conflict. One of the merging of states with common cores can never produce a shift/reduce conflict that was not present in one of the original states, because shift actions depend only on the core, not the lookahead. However, it is possible, that a merger will produce a reduce/reduce conflict.

Algorithm – Constructing LALR parsing table.

Input – An augmented grammar G' .

Output – The LALR parsing table functions action and goto for G' .

Method – (i) Construct $C = \{I_0, I_1, \dots, I_n\}$, the collection of sets of LR(1) items.

(ii) For each core in the set of LR(1) items, find all sets having that core, and replace these sets by their union.

(iii) Let $C' = \{J_0, J_1, \dots, J_m\}$ be the resulting sets of LR(1) items. The parsing actions for state i are constructed from J_i . If there is a parsing action conflict, the algorithm fails to produce parser and the grammar is said not to be LALR(1).

(iv) The GOTO table is constructed as follows. If J is the union of one or more sets of LR(1) items, that is $J = I_1 \cup I_2 \cup \dots \cup I_k$, then the cores of $\text{GOTO}(I_1, X), \text{GOTO}(I_2, X), \dots, \text{GOTO}(I_k, X)$ are the same, since I_1, I_2, \dots, I_k all have the same core. Let K be the union of all sets of items having the same core as $\text{GOTO}(I_1, X)$. Then $\text{GOTO}(J, X) = K$.

Table 2.20 LALR Parsing Table

STATE	ACTION			GOTO	
	c	d	\$	S	C
0	s36	s47		1	2
1			acc		
2	s36	s47			5
36	s36	s47			89
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		

The table formed is called LALR parsing table for G. If there are no parsing action conflicts, then the given grammar is said to be an LALR(1) grammar. The collection of sets of items constructed in step (iii) is called the LALR(1) collection.

The LALR ACTION and GOTO functions for the condensed set of items are shown in table 2.20.

Q.41. What are the different methods of constructing the LR parsing table from a given grammar? (R.G.P.V., June 2011)

Ans. Refer Q.35, Q.38 and Q.40.

Q.42. Explain parsing using ambiguous grammar.
Or

Can ambiguous grammars be used for LR parser construction? How? (R.G.P.V., Dec. 2005, June 2006)

Ans. There are certain types of ambiguous grammars that are useful in the specification and implementation of languages. An ambiguous grammar provides a shorter, more natural specification than any equivalent unambiguous grammar. Consider the following grammar for arithmetic expressions with operators + and *

$$E \rightarrow E + EE * E|(E)id$$

is the ambiguous because it does not specify the associativity or precedence of the operators + and *. The unambiguous grammar is

$$E \rightarrow E + T|T$$

$$T \rightarrow T * F|F$$

$$F \rightarrow (E)|id$$

The sets of LR(0) items are shown in fig. 2.14. The parsing action conflict in I_1 between accept and shift can be resolved by the SLR approach only \$ is in Follow (E), so acceptance is the unique action for input \$. On the other hand, + and * are the only inputs calling for a shift operation.

The conflict in I_7 between reduction by $E \rightarrow E + E$ and shift on + and * cannot be resolved by SLR method because + and * are in Follow (E). These conflicts can

$I_0: E' \rightarrow \cdot E$	$I_5: E \rightarrow E \cdot \cdot$
$E \rightarrow \cdot E + E$	$E \rightarrow E \cdot + E$
$E \rightarrow \cdot E * E$	$E \rightarrow E \cdot * E$
$E \rightarrow \cdot (E)$	$E \rightarrow (E) \cdot$
$E \rightarrow \cdot id$	$E \rightarrow id \cdot$
$I_1: E' \rightarrow E \cdot$	$I_6: E \rightarrow (E) \cdot$
$E \rightarrow E \cdot + E$	$E \rightarrow (E) \cdot + E$
$E \rightarrow E \cdot * E$	$E \rightarrow (E) \cdot * E$
$I_2: E \rightarrow (E) \cdot$	$I_7: E \rightarrow (E) \cdot + E$
$E \rightarrow (E) \cdot + E$	$E \rightarrow (E) \cdot * E$
$E \rightarrow (E) \cdot * E$	$E \rightarrow (E) \cdot (E)$
$E \rightarrow (E) \cdot (E)$	$E \rightarrow (E) \cdot id$
$E \rightarrow (E) \cdot id$	
$I_3: E \rightarrow id \cdot$	$I_8: E \rightarrow E \cdot + E$
$I_4: E \rightarrow E + \cdot E$	$E \rightarrow E + \cdot * E$
$E \rightarrow E + \cdot E$	$E \rightarrow E + \cdot (E)$
$E \rightarrow E + \cdot * E$	$E \rightarrow E + \cdot id$
$E \rightarrow E + \cdot (E)$	
$E \rightarrow E + \cdot id$	
	$I_9: E \rightarrow E \cdot (E)$

Fig. 2.14 Sets of LR(0) Items

be resolved using precedence and associativity information for + and *. If + is left-associative, the correct action is to reduce by $E \rightarrow E + E$. The * takes precedence over +, the action is shift action.

Similarly in state I_8 conflict occurs between reduction by $E \rightarrow E * E$ and shift on input + and *. Assuming that * is left associative and takes precedence over +, the state 8 appears on the top of the stack only when $E * E$ are the top three grammar symbols should have action reduce $E \rightarrow E * E$ on both + and * inputs.

The table 2.21 LR parsing table is obtained.

Table 2.21 Parsing Table for Grammar

STATE	ACTION						GOTO		
	id	+	*	()	\$	E	T	F
0	s3			s2			1		
1		s4	s5			acc			
2	s3			s2	r4		6		
3		r4	r4			r4			
4	s3			s2			8		
5	s3			s2	s9		8		
6		s4	s5		r1				
7		r1	s5		r2	r1			
8		r2	r2		r3	r2			
9		r3	r3			r3			

Consider another case of ambiguous grammar for conditional statements.
stmt \rightarrow if expr then stmt else stmt
| if expr then stmt
| other productions.

is ambiguous because it does not resolve the dangling else ambiguity. For simplicity rewrite the grammar, where i stands for if expr then, e for else, and a for "all other productions".

$$S' \rightarrow S$$

$$S \rightarrow iSeS/iS/a$$

The sets of LR(0) items for grammar are shown in fig. 2.15. The set of item I_4 yields a shift/reduce conflict, since $S \rightarrow iSeS$ calls for a shift of e while, since e is in FOLLOW (s), item $S \rightarrow iS \cdot$ calls for reduction by $S \rightarrow iS$ on e.

Translating back to the if .. then ... else terminology, given

If expr then stmt.

$I_0: S' \rightarrow \cdot S$	$I_3: S \rightarrow a \cdot$
$S \rightarrow \cdot iSeS$	$I_4: S \rightarrow iS \cdot eS$
$S \rightarrow \cdot iS$	$S \rightarrow iS \cdot$
$S \rightarrow \cdot a$	$I_5: S \rightarrow iSe \cdot S$
$I_1: S' \rightarrow S \cdot$	$S \rightarrow iSeS \cdot$
$I_2: S \rightarrow i \cdot SeS$	$S \rightarrow iS \cdot$
$S \rightarrow i \cdot S$	$S \rightarrow a \cdot$
$S \rightarrow iSe \cdot S$	$I_6: S \rightarrow iSeS \cdot$
$S \rightarrow iS \cdot$	
$S \rightarrow a \cdot$	

Fig. 2.15 LR(0) State for Augmented Grammar

on the stack and else as the first input symbol, should we shift else onto the stack (i.e., shift e) or reduce if expr then stmt to stmt (i.e., reduce by $S \rightarrow iS$). It is concluded that the conflict in I_4 should be resolved in favor of shift on input e. The parsing table constructed from the sets of items for abstract "dangling else" grammar is shown in table 2.22.

Table 2.22 Parsing Table for "dangling else" Grammar

STATE	ACTION				GOTO
	i	e	a	\$	
0	s_2		s_3		1
1				acc	
2	s_2		s_3		4
3		r_3		r_3	
4		s_5		r_2	
5	s_2		s_3		6
6		r_1		r_1	

For example, processing of input 'iiaea' is as follows –

Table 2.23 Processing of iiaea

S.No.	Stack	Input
(1)	0	iiaea \$
(2)	0i2	iaea \$
(3)	0i2i2	aea \$
(4)	0i2i2a3	ea \$
(5)	0i2i2S4	ea \$
(6)	0i2i2S4e5	a \$
(7)	0i2i2S4e5a3	\$
(8)	0i2i2S4e5s6	\$
(9)	0i2S4	\$
(10)	0s1	\$

Q.43. Describe the function of the LALR parser generator "Yacc".

Or

Write short note on "YACC". (R.G.P.V., June 2003, Dec. 2004, 2005, June 2007, Dec. 2007, 2008)

Or

Write short note on automatic parser generator.

(R.G.P.V., June 2005, 2008)

Ans. A parser generator can be used to facilitate the construction of the first end of a compiler. LALR parser generator 'yacc' stands for "yet another compiler", created by S.C. Johnson in early 1970's. Yacc is available as a command on the UNIX system, and has been used to implement hundreds of compilers.

A translator can be constructed using 'yacc'. First, a file, translate.y, containing a 'yacc' specification of the translator is prepared. The UNIX system command

yacc translate.y

transforms the file translate.y into a C program called y.tab.c using the LALR method. The program y.tab.c is a representation of an LALR parser written in C. By compiling y.tab.c along with the ly library that contains LR parsing program using the command.

cc y.tab.c -ly.

We obtain the desired object program a.out that performs the translation specified by the original 'yacc' program.

A 'yacc' source program has three parts –

declarations
%%
translation rules
%%
supporting C-routines.

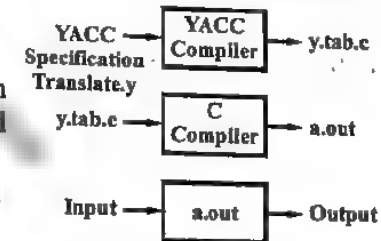


Fig. 2.16 Creating an I/O Translator with YACC

(i) **The Declarations Part** – There are two optional sections in the declarations part of a 'yacc' program. In the first section, ordinary C declarations like delimited by % { and % } are put. Here we place declarations of any temporaries used by the translation rules or procedures of the second and third sections. Declarations part also contain declarations of grammar tokens.

(ii) **The Translation Rules Part** – The translation rules are put after the first %% pair in the part of 'yacc' specification. Each rule consists of a grammar production and the associated semantic action. A 'yacc' semantic action is a sequence of C statements.

(iii) **The Supporting C-routines Part** – The third part of a 'yacc' specification consists of supporting C-routines. A lexical analyzer by the name yylex() must be provided. Other procedures such as error recovery routines may be added as necessary.

A Yacc desk calculator program derived from the given grammar is shown in fig. 2.17.

```

%{
#include <ctype.h>
%}
%token DIGIT
%%
line      :   expr '\n'      { printf("%d\n", $1); }
          ;
expr      :   expr '+' term  { $$ = $1 + $3; }
          |   term
          ;
  
```

```

term      : term '*' factor { $$ = $1 * $3; }
           | factor
factor     : '(' expre ')' { $$ = $2; }
           | DIGIT
%%
yylex( ) {
    int c;
    c = getchar( );
    if (isdigit(c)) {
        yylval = c-'0';
        return DIGIT;
    }
    return c;
}

```

Fig. 2.17 Yacc Specification of a Simple Desk Calculator

NUMERICAL PROBLEMS

Prob.21. For the following grammar –

$$E \rightarrow E + T | T$$

$$T \rightarrow T * F | F$$

$$F \rightarrow (E) | id$$

Construct the LR(0) canonical collection and also design the SLR parsing table. (R.G.P.V., Dec. 2011)

Sol. Refer to Q.25.

Prob.22. Consider the following grammar –

$$S \rightarrow SeS | iS | a$$

Construct the transition table and ACTION/GOTO table of the given grammar –

(i) Show that the given grammar is LR(0) or not

(ii) Show that the given grammar is SLR or not.

Sol. The augmented grammar will be –

$$S' \rightarrow S$$

$$S \rightarrow SeS$$

$$S \rightarrow iS$$

$$S \rightarrow a$$

The canonical collection sets of LR(0) items are computed as follows –

$$I_0 = \{$$

$$S' \rightarrow .S$$

$$S \rightarrow .SeS$$

$$S \rightarrow .iS$$

$$S \rightarrow .a$$

$$\}$$

$$\text{goto}(I_0, S) = \{ S' \rightarrow S.$$

$$S \rightarrow SeS$$

$$\} = I_1$$

$$\text{goto}(I_0, i) = \{ S \rightarrow i.S$$

$$S \rightarrow .SeS$$

$$S \rightarrow .iS$$

$$S \rightarrow .a$$

$$\} = I_2$$

$$\text{goto}(I_0, a) = \{ S \rightarrow a. \} = I_3$$

$$\text{goto}(I_1, e) = \{ S \rightarrow Se.S$$

$$S \rightarrow .SeS$$

$$S \rightarrow .iS$$

$$S \rightarrow .a$$

$$\} = I_4$$

$$\text{goto}(I_2, S) = \{ S \rightarrow iS.$$

$$S \rightarrow SeS$$

$$\} = I_5$$

$$\text{goto}(I_2, i) = I_2$$

$$\text{goto}(I_2, a) = I_3$$

$$\text{goto}(I_4, S) = \{ S \rightarrow SeS.$$

$$S \rightarrow SeS$$

$$\} = I_6$$

$$\text{goto}(I_4, i) = I_2$$

$$\text{goto}(I_4, a) = I_3$$

$$\text{goto}(I_5, e) = I_4$$

$$\text{goto}(I_6, e) = I_4$$

The transition diagram of the DFA is shown in fig. 2.18.

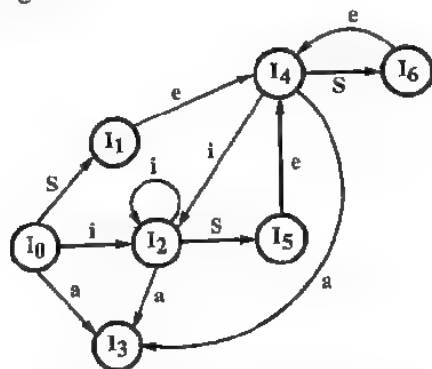


Fig. 2.18 DFA Transition Diagram

(i)

Table 2.24 LR(0) Parsing Table

STATE	ACTION TABLE				GOTO TABLE
	e	i	a	\$	
I ₀		S ₂	S ₃		1
I ₁	S ₄			Accept	
I ₂		S ₂	S ₃		5
I ₃	R ₃	R ₃	R ₃	R ₃	
I ₄		S ₂	S ₃		6
I ₅	S ₄ /R ₂	R ₂	R ₂	R ₂	
I ₆	S ₄ /R ₁	R ₁	R ₁	R ₁	

Since the ACTION/GOTO table shown in table 2.24 contains multiple entries, so the given grammar is not LR(0).

(ii)

Table 2.25 SLR Parsing Table

STATE	ACTION TABLE				GOTO TABLE
	e	i	a	\$	
I ₀		S ₂	S ₃		1
I ₁	S ₄			Accept	
I ₂		S ₂	S ₃		5
I ₃	R ₃			R ₃	
I ₄		S ₂	S ₃		6
I ₅	S ₄ /R ₂			R ₂	
I ₆	S ₄ /R ₁			R ₁	

Since the ACTION/GOTO table shown in table 2.25 contains multiple entries, so the given grammar is not SLR.

Prob.23. Construct the collection of LR(0) item sets and draw the goto graph for the following grammar –

$$S \rightarrow SS \mid a \mid \epsilon$$

Indicate the conflicts (if any) in the various states of the SLR parser.

(R.G.P.V., Dec. 2008)

Sol. The augmented grammar will be –

$$S' \rightarrow S$$

$$S \rightarrow SS$$

$$S \rightarrow a$$

$$S \rightarrow \epsilon$$

The canonical collection sets of LR(0) items are computed as follows –

$$I_0 = \text{closure}(\{S' \rightarrow \cdot S\}) = \{S' \rightarrow \cdot S$$

$$S \rightarrow \cdot SS$$

$$S \rightarrow \cdot a$$

$$S \rightarrow \cdot$$

$$\}$$

$$\text{goto}(I_0, S) = \text{closure}(\{S' \rightarrow S \cdot\})$$

$$= \{S' \rightarrow S \cdot$$

$$S \rightarrow S \cdot S$$

$$S \rightarrow \cdot SS$$

$$S \rightarrow \cdot a$$

$$S \rightarrow \cdot$$

$$\} = I_1$$

$$\text{goto}(I_0, a) = \{S \rightarrow a \cdot\} = I_2$$

$$\text{goto}(I_1, S) = \{S \rightarrow SS \cdot$$

$$S \rightarrow S \cdot S$$

$$S \rightarrow \cdot SS$$

$$S \rightarrow \cdot a$$

$$S \rightarrow \cdot$$

$$\} = I_3$$

$$\text{goto}(I_1, a) = \{S \rightarrow a \cdot\} = \text{same as of } I_2$$

$$\text{goto}(I_3, S) = \{S \rightarrow SS \cdot$$

$$S \rightarrow S \cdot S$$

$$S \rightarrow \cdot SS$$

$$S \rightarrow \cdot a$$

$$S \rightarrow \cdot$$

$$\} = \text{same as of } I_3$$

$$\text{goto}(I_3, a) = I_2$$

Yes, there is a conflict in the I₁ state of the SLR parser.

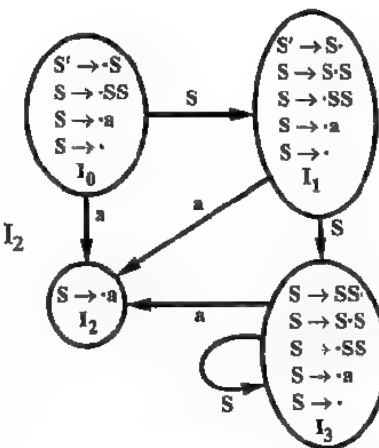


Fig. 2.19 The Goto Graph

Prob.24. Show that the following grammar –

$$S \rightarrow Aa \mid bAc \mid dc \mid bda$$
$$A \rightarrow d$$

is **LALR (1)** but not **SLR (1)**.

(R.G.P.V., June 2005, Dec. 2006)

Sol. The augmented grammar is –

$$S' \rightarrow S$$
$$S \rightarrow Aa$$
$$S \rightarrow bAc$$

$S \rightarrow dc$

$$S \rightarrow bda$$
$$A \rightarrow d$$

The canonical collection of sets of LR(1) items are –

$$I_0 = \{S' \rightarrow .S, \$$$
$$S \rightarrow .Aa, \$$$
$$S \rightarrow .bAc, \$$$
$$S \rightarrow .dc, \$$$

$S \rightarrow .bda, \$$

$$A \rightarrow .d, a$$

✓

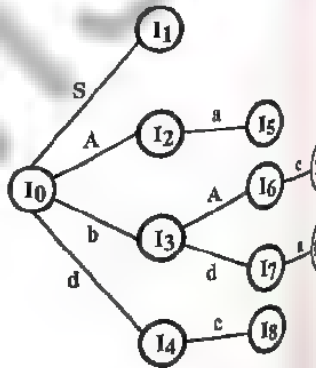
$$\text{goto}(I_0, S) = \{S' \rightarrow S., \$\} = I_1$$
$$\text{goto}(I_0, A) = \{S \rightarrow A.a, \$\} = I_2$$
$$\text{goto}(I_0, b) = \{S \rightarrow b.Ac, \$$$
$$S \rightarrow b.da, \$$$
$$A \rightarrow .d, c$$
$$\} = \mathbf{I}_3$$
$$\text{goto}(I_0, d) = \{S \rightarrow d.c, \$$$
$$A \rightarrow d, a$$
$$\} = I_4$$
$$\text{goto}(I_2, a) = \{S \rightarrow Aa., \$\} = I_5$$
$$\text{goto}(I_3, A) = \{S \rightarrow bA.c, \$\} = I_6$$
$$\text{goto}(I_3, d) = \{S \rightarrow bd.a, \$$$
$$A \rightarrow d, c$$
$$\} = I_7$$
$$\text{goto}(I_4, c) = \{S \rightarrow dc., \$\} = I_6$$
$$\text{goto}(I_6, c) = \{S \rightarrow bAc., \$\} = I_9$$
$$\text{goto}(I_7, a) = \{S \rightarrow bda., \$\} = I_{10}$$


Fig. 2.20 DFA

There are no sets of LR(1) items in the canonical collection that have identical LR(0) – part items and that differ only in their lookaheads. So, the LALR(1) parsing table for the given grammar is given in table 2.26. This table contains no multiple defined entries, hence, the given grammar is LALR (1).

Table 2.26 LALR(1) Parsing Table

STATE	ACTION TABLE					GOTO TABLE	
	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	<i>\$</i>	<i>S</i>	<i>A</i>
I_0		S_3		S_4		1	2
I_1					Accept		
I_2	S_5						
I_3				S_7			6
I_4	R_5		S_8				
I_5					R_1		
I_6			S_9				
I_7	S_{10}		R_5				
I_8					R_3		
I_9					R_2		
I_{10}					R_4		

The SLR(1) parsing for the given grammar is given in table 2.27.

Table 2.27 SLR(1) Parsing Table

STATE	ACTION TABLE					GOTO TABLE	
	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	<i>\$</i>	<i>S</i>	<i>A</i>
I_0			S_3	S_4		1	2
I_1					Accept		
I_2	S_5						
I_3				S_7			6
I_4	R_5		S_8/R_5				
I_5					R_1		
I_6			S_9				
I_7	S_{10}/R_5		R_5				
I_8					R_3		
I_9					R_2		
I_{10}					R_4		

Since ACTION/GOTO table of SLR(1) contains multiple entries. So it is not SLR(1).

Since the ACTION/GOTO table shown in table 2.28 does not contain multiple entries, so the given grammar is LR(1).

There is only one set of LR(1) items in the canonical collection that have identical LR(0) part items and that differ only in their lookaheads i.e. I_5 and I_9 . So, the LALR(1) parsing table for the grammar is given in table 2.29.

Table 2.29 LALR(1) Parsing Table

STATE	ACTION TABLE					GOTO TABLE		
	a	b	c	d	\$	S	A	R
I_0		S_3		$S_{5,9}$		1	2	4
I_1					Accept			
I_2	S_6							
I_3			S_7	$S_{5,9}$			7	8
I_4			S_{10}					
$I_{5,9}$	R_5/R_6		R_5/R_6					
I_6					R_1			
I_7			S_{11}					
I_8	S_{12}							
I_{10}					R_3			
I_{11}					R_2			
I_{12}					R_4			

Since the ACTION/GOTO table shown in table 2.29 contains multiple entries, so the given grammar is not LALR(1).

Prob.26. Construct LALR items for the grammar below -

$$E \rightarrow E + T \mid T$$

$$T \rightarrow T * F \mid F$$

$$F \rightarrow (E) \mid id$$

Sol. The augmented grammar is

$$E' \rightarrow E$$

$$E \rightarrow E + T \mid T$$

$$T \rightarrow T * F \mid F$$

$$F \rightarrow (E) \mid id$$

(R.G.P.V., June 2013)

The canonical collection of sets of LR(1) items are computed as follows -

$$I_0 = \{ E' \rightarrow .E, \$ \\ E \rightarrow .E + T, \$ \mid + \\ E \rightarrow .T, \$ \mid + \\ T \rightarrow .T * F, \$ \mid + \mid * \\ T \rightarrow .F, \$ \mid + \mid * \\ F \rightarrow .(E), \$ \mid + \mid * \\ F \rightarrow .id, \$ \mid + \mid * \}$$

$$\text{goto}(I_0, E) = \{ E' \rightarrow E., \$ \\ E \rightarrow E. + T, \$ \mid + \}$$

$$\} = I_1$$

$$\text{goto}(I_0, T) = \{ E \rightarrow T., \$ \mid + \\ T \rightarrow T. * F, \$ \mid + \mid * \}$$

$$\} = I_2$$

$$\text{goto}(I_0, F) = \{ T \rightarrow F., \$ \mid + \mid * \} = I_3$$

$$\text{goto}(I_0, () = \{ F \rightarrow (.E), \$ \mid + \mid * \\ E \rightarrow .E + T,) \mid + \\ E \rightarrow .T,) \mid + \\ T \rightarrow .T * F,) \mid + \mid * \\ T \rightarrow .F,) \mid + \mid * \\ F \rightarrow .(E,) \mid + \mid * \\ F \rightarrow .id,) \mid + \mid * \}$$

$$\} = I_4$$

$$\text{goto}(I_0, id) = \{ F \rightarrow id., \$ \mid + \mid * \} = I_5$$

$$\text{goto}(I_1, +) = \{ E \rightarrow E + .T, \$ \mid + \\ T \rightarrow .T * F, \$ \mid + \mid * \\ T \rightarrow .F, \$ \mid + \mid * \\ F \rightarrow .(E), \$ \mid + \mid * \\ F \rightarrow .id, \$ \mid + \mid * \}$$

$$\} = I_6$$

$$\text{goto}(I_2, *) = \{ T \rightarrow T * .F, \$ \mid + \mid * \\ F \rightarrow .(E), \$ \mid + \mid * \\ F \rightarrow .id, \$ \mid + \mid * \}$$

$$\} = I_7$$

$$\text{goto}(I_4, E) = \{ F \rightarrow (E.), \$ \mid + \mid * \\ E \rightarrow E. + T,) \mid + \}$$

$$\} = I_8$$

$$\text{goto}(I_4, T) = \{ E \rightarrow T.,) \mid + \\ T \rightarrow T. * F,) \mid + \mid * \}$$

$$\} = I_9$$

Prob.27. Consider the following grammar –

$$E \rightarrow E + T \mid T$$

$$T \rightarrow TF \mid F$$

$$F \rightarrow F^* \mid a \mid b$$

- (i) Construct the SLR parsing table for this grammar.
(ii) Construct the LALR parsing table.

(R.G.P.V., Nov. 2018)

Sol. (i) The augmented grammar is

$$E' \rightarrow E$$

$$E \rightarrow E + T \mid T$$

$$T \rightarrow TF \mid F$$

$$F \rightarrow F^* \mid a \mid b$$

The canonical collection of sets of items are computed as follows–

$$I_0 = \{ E' \rightarrow .E$$

$$E \rightarrow .E + T$$

$$E \rightarrow .T$$

$$T \rightarrow .TF$$

$$T \rightarrow .F$$

$$F \rightarrow .F^*$$

$$F \rightarrow .a$$

$$F \rightarrow .b$$

$$\text{goto}(I_0, E) = \{ E' \rightarrow E. \\ E \rightarrow E. + T$$

$$\} = I_1$$

$$\text{goto}(I_0, T) = \{ E \rightarrow T.$$

$$T \rightarrow T.F$$

$$F \rightarrow F^*$$

$$F \rightarrow .a$$

$$F \rightarrow .b$$

$$\} = I_2$$

$$\text{goto}(I_0, F) = \{ T \rightarrow F.$$

$$F \rightarrow F^*$$

$$F \rightarrow .a, \mid \$ \mid + \mid *$$

$$F \rightarrow .b, \mid \$ \mid + \mid *$$

$$\} = I_3$$

$$\text{goto}(I_0, a) = \{ F \rightarrow a.$$

$$\} = I_4$$

$$\text{goto}(I_0, b) = \{ F \rightarrow b.$$

$$\} = I_5$$

$$\text{goto}(I_1, +) = \{ E \rightarrow E + .T$$

$$T \rightarrow .TF$$

$$T \rightarrow .F$$

$$T \rightarrow .F$$

$$F \rightarrow .a$$

$$F \rightarrow .b$$

$$\} = I_6$$

$$\text{goto}(I_2, F) = \{ T \rightarrow TF.$$

$$F \rightarrow F^*$$

$$\} = I_7$$

$$\text{goto}(I_2, a) = I_4$$

$$\text{goto}(I_2, b) = I_5$$

$$\text{goto}(I_3, *) = \{ F \rightarrow F^*.$$

$$\} = I_8$$

$$\text{goto}(I_6, T) = \{ E \rightarrow E + T.$$

$$T \rightarrow T.F$$

$$F \rightarrow F^*$$

$$F \rightarrow .a$$

$$F \rightarrow .b$$

$$\} = I_9$$

$$\text{goto}(I_6, F) = \{ T \rightarrow F.$$

$$F \rightarrow F^*$$

$$\} = I_{10}$$

$$\text{goto}(I_6, a) = I_4$$

$$\text{goto}(I_6, b) = I_5$$

$$\text{goto}(I_7, *) = I_8$$

$$\text{goto}(I_9, F) = I_7$$

$$\text{goto}(I_9, a) = I_4$$

$$\text{goto}(I_9, b) = I_5$$

$$\text{goto}(I_{10}, *) = I_8$$

Table 2.31 SLR Parsing Table

STATE	ACTION TABLE					GOTO TABLE		
	a	b	+	*	\$	E	F	T
I ₀	S ₄	S ₅				1	3	2
I ₁			S ₆		Accept			
I ₂	S ₄	S ₅			R ₂		7	
I ₃			R ₄	S ₈	R ₄			
I ₄	R ₆	R ₆	R ₆		R ₆			
I ₅	R ₇	R ₇	R ₇		R ₇			
I ₆	S ₄	S ₅					10	9
I ₇			R ₃	S ₈	R ₃			
I ₈	R ₅	R ₅	R ₅		R ₅			
I ₉	S ₄	S ₅			R ₁		7	
I ₁₀			R ₄	S ₈	R ₄			

(ii) The augmented grammar is

$$\begin{aligned} &\{ \\ &\quad E' \rightarrow E, \\ &\quad E \rightarrow E + T \mid T \\ &\quad T \rightarrow TF \mid F \\ &\quad F \rightarrow F * \mid a \mid b \end{aligned}$$

The cononical set of items are computed as follows –

$$\begin{aligned} I_0 = \{ & \\ &\quad E' \rightarrow .E, \$ \\ &\quad E \rightarrow .E + T, \$ \\ &\quad E \rightarrow .T, \$ \\ &\quad T \rightarrow .TF, \$ \\ &\quad T \rightarrow .F, \$ \\ &\quad F \rightarrow .F *, \$ \\ &\quad F \rightarrow .a, \$ \\ &\quad F \rightarrow .b, \$ \\ &\} = I_0 \end{aligned}$$

Now we shall apply goto operation

goto (I_0 , E)

$$\begin{aligned} &\{ \\ &\quad E' \rightarrow E., \$ \\ &\quad E \rightarrow E. + T, \$ \\ &\} = I_1 \end{aligned}$$

goto (I_0 , T)

$$\begin{aligned} &\{ \\ &\quad E \rightarrow T., \$ \\ &\quad T \rightarrow T.F, \$ \\ &\quad F \rightarrow .F *, \$ \\ &\quad F \rightarrow .a, \$ \\ &\quad F \rightarrow .b, \$ \\ &\} = I_2 \end{aligned}$$

goto (I_0 , F)

$$\begin{aligned} &\{ \\ &\quad T \rightarrow F., \$ \\ &\quad F \rightarrow F. *, \$ \\ &\} = I_3 \end{aligned}$$

goto (I_0 , a)

$$\begin{aligned} &\{ \\ &\quad F \rightarrow a., \$ \\ &\} = I_4 \end{aligned}$$

goto (I_0 , b)

$$\begin{aligned} &\{ \\ &\quad F \rightarrow b., \$ \\ &\} = I_5 \end{aligned}$$

goto (I_1 , T)

$$\begin{aligned} &\{ \\ &\quad E \rightarrow E + .T, \$ \\ &\quad T \rightarrow .TF, \$ \\ &\quad T \rightarrow .F, \$ \\ &\quad F \rightarrow .F *, \$ \\ &\quad F \rightarrow .a, \$ \\ &\quad F \rightarrow .b, \$ \\ &\} = I_6 \end{aligned}$$

goto (I_2 , F)

$$\begin{aligned} &\{ \\ &\quad T \rightarrow TF, \$ \\ &\quad F \rightarrow F., \$ \\ &\} = I_7 \end{aligned}$$

goto (I_2 , a) = I_4

goto (I_2 , b) = I_5

goto (I_3 , *)

$$\begin{aligned} &\{ \\ &\quad F \rightarrow F *, \$ \\ &\} = I_8 \end{aligned}$$

goto (I_6 , T)

$$\begin{aligned} &\{ \\ &\quad E \rightarrow E + T., \$ \\ &\quad T \rightarrow T.F, \$ \\ &\quad F \rightarrow .F *, \$ \\ &\quad F \rightarrow .a, \$ \\ &\quad F \rightarrow .b, \$ \\ &\} = I_9 \end{aligned}$$

goto (I_6 , F) = I_3

goto (I_6 , a) = I_4

goto (I_6 , b) = I_5

goto (I_7 , *) = I_8

goto (I_9 , F) = I_7

goto (I_9 , a) = I_4

goto (I_9 , b) = I_5

Table 2.32 LALR (1) Parsing Table

STATE	ACTION TABLE					GOTO TABLE		
	a	b	+	*	\$	E	F	T
I ₀	S ₄	S ₅				1	3	2
I ₁			S ₆		Accept			
I ₂	S ₄	S ₅			R ₂		7	
I ₃				S ₈	R ₄			
I ₄					R ₆			
I ₅					R ₇			
I ₆	S ₄	S ₅					3	9
I ₇				S ₈	R ₃			
I ₈					R ₅			
I ₉	S ₄	S ₅			R ₁		7	

SYNTAX DIRECTED DEFINITIONS – CONSTRUCTION OF SYNTAX TREES, BOTTOM UP EVALUATION OF S-ATTRIBUTED DEFINITION, L-ATTRIBUTE DEFINITION, TOP DOWN TRANSLATION, BOTTOM UP EVALUATION OF INHERITED ATTRIBUTES, RECURSIVE EVALUATION, ANALYSIS OF SYNTAX DIRECTED DEFINITION

Q.44. Write a short note on syntax directed translation.

(R.G.P.V., June 2008, 2011)

Or

Explain the syntax directed definition for constructing syntax tree for an arithmetic expression. Also explain what is annotated parse tree.

(R.G.P.V., Dec. 2013)

Or

What is syntax directed translation? Why are they important?

(R.G.P.V., Dec. 2014)

Or

Define syntax directed definition.

(R.G.P.V., Dec. 2018)

Ans. Syntax directed definition are high level specifications for translators. They hide many implementation details and free the user from having to specify explicitly the order in which translation takes place. Translation schemes indicate the order in which semantic rules are to be evaluated, so they allow some implementation details to be shown. A syntax directed definition is

generalization of context free grammar in which each grammar symbol has an associated set of attributes. Syntax directed translation provides a convenient framework to specify compilation actions. It has three components – a source language grammar, a set of attributes, and a set of semantic actions. A parser parses a source program according to the source language grammar and constructs a parse tree in which each node represents a grammar symbol. Attribute describes some compile time information associated with the symbol e.g., the type of a variable or expression, the address of its memory location, the sequence of instructions generated for a construct, etc.

A semantic action assigns an appropriate value to an attribute of a grammar symbol. It is associated with a grammar production. It may also contribute to translation of a program by performing symbol table building, memory allocation or code generation etc. A semantic action is enclosed between {.....}. Attributes can be partitioned into two subsets called the synthesized and inherited attributes. The value of a synthesized attribute at a node is computed from the values of attributes at the children of the node in the parse tree. The value of an inherited attribute is computed from the values of attribute at the siblings and parent of that node. Semantic rules set up dependencies between attributes that will be represented by a graph. From the dependency graph, evaluation order for semantic rules are derived. Evaluation of a semantic rules defines the values of attributes at each node is called an annotated parse tree. The process of computing the attribute values at the node is called annotating or decorating the parse tree.

For every grammar symbol appearing in the parse tree a value is associated and they together form the translation of that symbol. A parse tree thus modified denoted a syntax directed translation scheme. It has entry such as X.VAL or X.TRUE where X is the grammar symbol and X.VAL is the value that takes when actual evaluation occurs. If we have a production with several instances of the same symbol on the right they are distinguished with separate superscripts.

e.g., $E \rightarrow E^{(1)} + E^{(2)}$ actually get converted as

$$E.VAL := E^{(1)}.VAL + E^{(2)}.VAL$$

which states that translation E.VAL on L.H.S. is determined by adding together the translation associated with the E's on the right side of the production.

This translation scheme allows subroutine or "semantic actions" to be attached to the production of a context free grammar. These subroutines generate intermediate code when called at appropriate times by a parser for that grammar.

Let us consider how the semantic actions defines the value of translation. A small grammar with its production is given below with the semantic actions

$$E \rightarrow E^{(1)} + E^{(2)} \{ E.VAL := E^{(1)}.VAL + E^{(2)}.VAL \}$$

$$E \rightarrow \text{digit} \{ E.VAL := \text{digit} \}$$

where digit stands for any digit between 0 to 9. If suppose we have a input string $1 + 2 + 3$. A parse tree for this expression is given in fig. 2.22 and also another tree using the syntax translation is drawn in fig. 2.23.

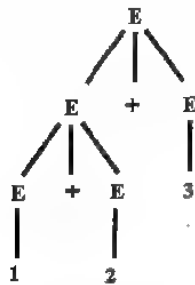


Fig. 2.22 Parse Tree for Expression $1 + 2 + 3$

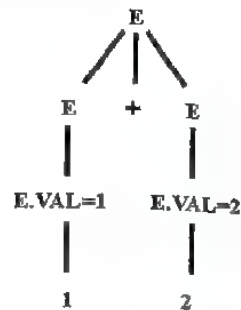


Fig. 2.23 Subtree with Computed Translation

Considering the bottom leftmost E. This node corresponding to a use of the production $E \rightarrow 1$. The corresponding semantic action sets $E.VAL = 1$ thus we can associate the value 1 with the translation $E.VAL$ at the bottom leftmost E. Similarly we can associate the value 2 with the translation $E.VAL$ at the right sibling of this node.

The value of $E.VAL$ at the root is calculated using the semantic rule

$$E.VAL := E^{(1)}.VAL + E^{(2)}.VAL$$

In applying this rule we substitute the value of $E.VAL$ at the bottom leftmost E for $E^{(1)}.VAL$ and the value of $E.VAL$ at its right sibling E for $E^{(2)}.VAL$.

In fig. 2.24 the process of evaluation is $1 + 2 = 3$ and then $3 + 3 = 6$. The semantic action associated with grammar production are put to use. First the lower subtree is evaluated so as to get its root a value $1 + 2 = 3$ and then the upper subtree is evaluated to get finally 6.

The output defined is independent of the kind of parser used to construct the parse tree. Such scheme provides a method for describing an input-output mapping and that description is independent of any implementation. Such schemes are very easy to modify. New production with semantic actions can be easily added or deleted.

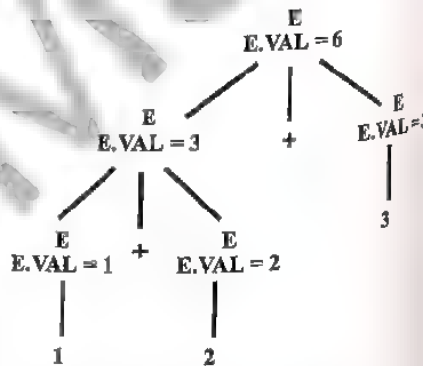


Fig. 2.24 Complete Parse Tree

Q.45. Assuming suitable syntax directed definition, construct a syntax tree for the expression $a - 4 + e$. (R.G.P.V., Dec. 2017)

Ans. A parse tree represented by dotted lines in fig. 2.25 shows the construction of a syntax tree for the expression $a - 4 + e$. The parse tree nodes use the synthesized attribute "p" to hold a pointer to the node of syntax tree for the expression depicted through nonterminals E and T. Attribute T.p to be a pointer to a new leaf for an identifier and a number is defined by the semantic rules which are connected with the productions $T \rightarrow id$ and $T \rightarrow num$. Lexical analyzer with the tokens *id* and *num* returns attributes *id.entry* and *num.val* which are lexical values. In fig. 2.25, when an expression E is a single term, it is equivalent to the use of the production $E \rightarrow T$. The value of attribute T.p is obtained by the attribute E.p. Previous rules have set $E_1.p$ and T.p to be pointers to the leaves for 'a' and '4' respectively. When the semantic rule $E.p := mknode('-', E_1.p, T.p)$ connected with the production $E \rightarrow E_1 - T$ is invoked. In interpreting fig. 2.25, it is essential to know that the lower tree, which is developed from records is a real syntax tree that has output, but the dotted tree above is the parse tree develop only in a figurative sense.

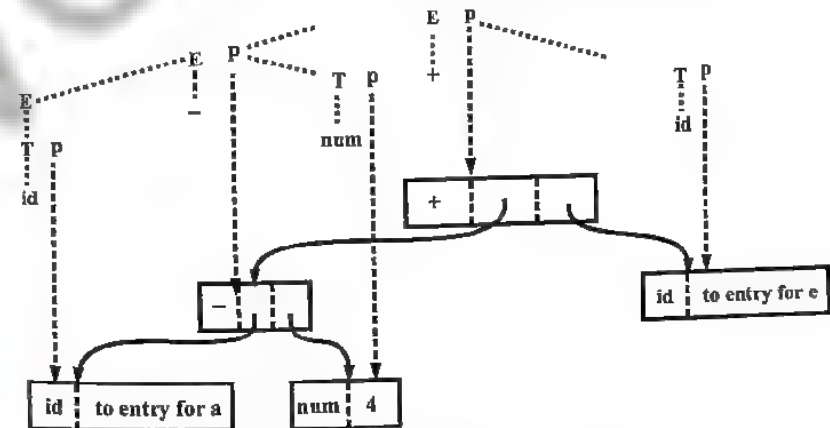


Fig. 2.25 Syntax Tree Construction for $a - 4 + e$

Q.46. What is meant by syntax directed translation? Explain. Give the parse tree and translations for the expression $23 * 5 + 4$ according to the syntax directed translation scheme. (R.G.P.V., Dec. 2009)

Ans. Syntax Directed Translation – Refer to Q.44.

For the input expression $23 * 5 + 4$, the program is to produce the value 119. To design such a translator, we must first write a grammar to describe the inputs. We use the nonterminals E for expression and I for integer. The

productions are –

$$\begin{aligned} E &\rightarrow E + E \\ E &\rightarrow E * E \\ E &\rightarrow I \\ I &\rightarrow I \text{ digit} \\ I &\rightarrow \text{digit} \end{aligned}$$

The terminals are +, * and digit.

Now, we add the semantic actions to the productions. With each of the nonterminals E and I we associate one integer-valued translation, called E.val and I.val, respectively, which denotes the numerical value of the expression or integer represented by a node of the parse tree labeled E or I. With the terminal digit we associate the translation LEXVAL, which we assume is the second component of the pair (digit, LEXVAL) returned by the lexical analyzer when a token of type digit is found. Semantic action for grammar is shown in fig. 2.26.

Production	Semantic Action
$E \rightarrow E^{(1)} + E^{(2)}$	$\{E.VAL := E^{(1)}.VAL + E^{(2)}.VAL\}$
$E \rightarrow E^{(1)} * E^{(2)}$	$\{E.VAL := E^{(1)}.VAL * E^{(2)}.VAL\}$
$E \rightarrow I$	$\{E.VAL := I.VAL\}$
$I \rightarrow I^{(1)} \text{ digit}$	$\{I.VAL := 10 * I^{(1)}.VAL + \text{LEXVAL}\}$
$I \rightarrow \text{digit}$	$\{I.VAL := \text{LEXVAL}\}$

Fig. 2.26 Syntax Directed Translation Scheme

Using syntax-directed translation scheme, the input $23 + 5 * 4$ would have the parse tree and translations shown in fig. 2.27.

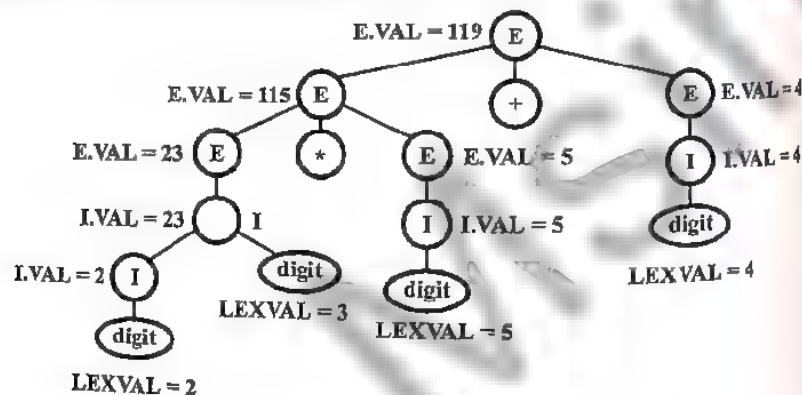


Fig. 2.27 Parse Tree with Translations

Q.47. What is meant by syntax directed translation? Explain. Give the parse tree and translations for the expression $25 * 5 + 4$ according to the syntax directed translation scheme.

(R.G.P.V., June 2016)

Ans. Syntax Directed Translation – Refer to Q.44.

Parse Tree and Translation – Similar as Q.46.

Q.48. What is a translation scheme? How is it different from a syntax directed definition? Illustrate the order of execution of semantic actions in a translation scheme.

(R.G.P.V., Dec. 2015)

Ans. For specifying translation during parsing, a translation scheme is a useful notation. A translation scheme permits us to specify the semantic rules expressing the relationship between attributes as well as the order of evaluation of attributes. A translation scheme is a context free grammar in which program fragments called semantic actions are embedded within the right side of productions. The position at which an action is to be executed is shown by enclosing it between braces and writing it within the right side of production.

A translation scheme produces an output for each sentence x generated by the underlying grammar by executing the actions in the order they appear during a depth first traversal of a parse tree for x . When drawing a parse tree for a translation scheme, we indicate an action by constructing for it an extra child, connected by a dashed line to the node for its production. Translation schemes are designed to ensure that the semantic actions refer to attributes whose values are already computed by a previous semantic action or by the lexical analyzer.

In a syntax directed definition, semantic actions associated with a grammar production need not be performed in any specific order, whereas in a translation scheme the actions must be performed in the same order in which they are specified.

Refer to Q.44.

Q.49. What are advantages of syntax directed translation?

Ans. Syntax directed translation is a scheme that indicates the order in which semantic rules are to be evaluated. The main advantage of SDT is that it helps in deciding evaluation order. The evaluation of semantic actions associated with SDT may generate code, save information in symbol table, or may issue error message.

Q.50. Explain the types of attribute used in syntax directed translation.

Or

Differentiate between inherited attribute and synthesized attribute with example.

(R.G.P.V., June 2016)

Ans. In a syntax directed translation, each production $A \rightarrow \alpha$ has associated with it a set of semantic rules of form $b := f(c_1, c_2, \dots, c_k)$ where f is a function, and

- b is a synthesized attribute of A and c_1, c_2, c_k are attributes belonging to the grammar symbols on the production, or
- b is an inherited attribute of one of the grammar symbols on the right side of the production, and c_1, c_2, c_k are attributes belonging to the grammar symbols of the production.

(i) **Synthesized Attributes** – An attribute is said to be synthesized if its value at a parse-tree node is determined from attributes values at the children of the node. Synthesized attributes are used extensively in practice. A syntax directed definition that uses synthesized attributes is said to be an S-attributed definition. An S-attributed definition parse tree can always be annotated by evaluating the semantic rules for the attributes at each node bottom up, from the levels to the root.

(ii) **Inherited Attributes** – An inherited attribute is one whose value at a node in a parse tree is defined in terms of attribute at the parent and/or siblings of that node. Inherited attributes are used for expressing the dependence of a programming language construct on the context in which it appears. It is always possible to rewrite a syntax directed definition to use only synthesized attributes, it is often more natural to use syntax directed definition with inherited attributes. An inherited attribute distributes type information to the various identifiers in a declaration.

Q.51. Differentiate between synthesized translation and inherited translations.

(R.G.P.V., Dec. 2005, June 2008, 2011)

Ans. Synthesized translation defines the value of the translation of the nonterminal on the left side of production as a function of the translations of the nonterminals on the right side. Such a translation is called a *synthesized* translation.

Consider the following production and action

$$A \rightarrow XYZ \{Y.VAL := 2 * A.VAL\}$$

Here the translation of a nonterminal on the right side of the production is defined in terms of a translation of the nonterminal on the left. Such a translation is called an *inherited* translation.

We shall use synthesized attribute exclusively. They are more natural than inherited translation for mapping most programming language constructs into intermediate code, and they can be implemented simply in conjunction with bottom-up or top-down parser.

Q.52. How can syntax trees be constructed? Explain in brief. Explain implementation of syntax directed translator.

Write a brief note on syntax tree. (R.G.P.V., Dec. 2005, June 2008)

Or

What do you mean by syntax tree?

Or

Write short note on syntax trees.

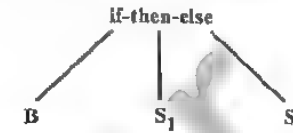
(R.G.P.V., June 2010)

(R.G.P.V., Dec. 2010)

Ans. Syntax directed definitions can be used to specify the construction of syntax trees and other graphical representations of language constructs.

The use of syntax trees as an intermediate representation allows translation to be decoupled from parsing. A syntax (abstract) tree is a condensed form

of parse tree useful for representing language constructs. The production $S \rightarrow \text{if } B \text{ then } S_1 \text{ else } S_2$ might appear in a syntax tree as –



In a syntax tree, operators and keywords do not appear as leaves, but associated with the interior node that would be the parent of those leaves in the parse tree. Syntax directed translation can be based on syntax trees as well as parse trees.

A syntax directed translation scheme provides a method for describing an input-output mapping and that description is independent of any implementation. It is easy to modify. After writing a syntax directed translation scheme it is converted into a program that implements the input output mapping. One way to implement a syntax directed translator is to use extra fields in the parser stack entries corresponding translations. If a grammar involved is LR grammar then the convenient shift reduce technique can be used for doing a bottom up parsing. For example, the arithmetic expression grammar is given as follows – where S is a nonterminal, E stands for expression and I for integer. The productions are

- | | |
|------------------------------------|--|
| (i) $S \rightarrow E\$$ | (ii) $E \rightarrow E + E$ |
| (iii) $E \rightarrow E * E$ | (iv) $E \rightarrow (E)$ |
| (v) $E \rightarrow I$ | (vi) $I \rightarrow \text{Idigit}$ |
| (vii) $I \rightarrow \text{digit}$ | (viii) $\text{digit} \rightarrow 0/1/2/3 \dots /9$ |

The semantic actions are added to the productions, with each of the nonterminals E and I , one integer-valued translation called $E.VAL$ and $I.VAL$ is associated. Which denoted the numerical value of the expression or integer represented by a node of the parse tree labeled E or I . The set of semantic action is generated for each and every production as the first step. The set of semantic actions are –

S.No.	Productions	Semantic Actions
(i)	$S \rightarrow E\$$	{ print $E.VAL$ }
(ii)	$E \rightarrow E^{(1)} + E^{(2)}$	{ $E.VAL := E^{(1)}.VAL + E^{(2)}.VAL$ }
(iii)	$E \rightarrow E^{(1)} * E^{(2)}$	{ $E.VAL := E^{(1)}.VAL * E^{(2)}.VAL$ }
(iv)	$E \rightarrow (E^{(1)})$	{ $E.VAL := E^{(1)}.VAL$ }
(v)	$E \rightarrow I$	{ $E.VAL := I.VAL$ }
(vi)	$I \rightarrow I^{(1)}\text{digit}$	{ $I.VAL := 10 * I^{(1)}.VAL + \text{LEXVAL}$ }
(vii)	$I \rightarrow \text{digit}$	{ $I.VAL := \text{LEXVAL}$ }

The last production only indicates different assignment of digit hence require no semantic action specification. The semantic action for the seventh production involves a term LEXVAL i.e., lexical values, which means any of

the value associated with digit (0 to 9) as indicated by the expression. Here digit is a number token and its attribute will be the lexical value. The complete parse tree with translation can be shown in fig. 2.28.

The parsing of a given parse tree using stack shift reduce technique can be easily done. Table 2.33 shows the sequence of moves made by the parser on input 23*5+4\$. The column STATE indicates the status of the stack and the column VAL gives valuation result at any particular state. Input and production columns represent the input applied and production used for evaluation respectively.

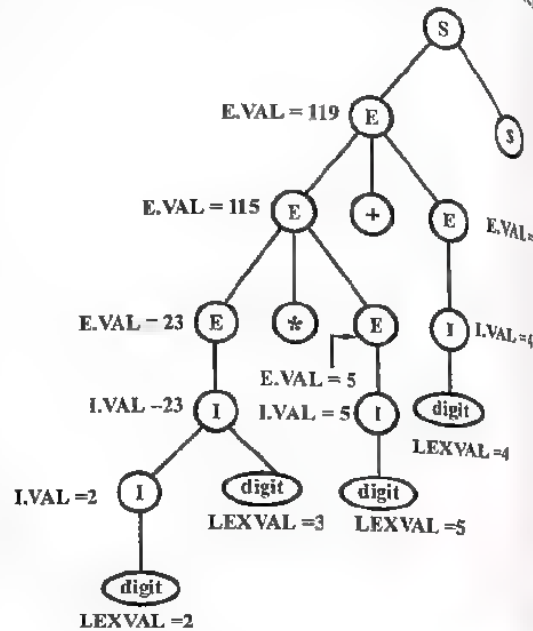


Fig. 2.28 Parse Tree with Translations

Table 2.33 Sequence of Moves

S.No.	Input	State	VAL	Production Used
(1)	23*5+4\$	-	-	
(2)	3*5+4\$	2	-	
(3)	3*5+4\$	I	2	$I \rightarrow \text{digit}$
(4)	*5+4\$	I3	2 -	
(5)	*5+4\$	I	(23)	$I \rightarrow \text{Idigit}$
(6)	*5+4\$	E	(23)	$E \rightarrow I$
(7)	5+4\$	E*	(23) -	
(8)	+4\$	E*5	(23) --	
(9)	+4\$	E*I	(23) -5	$I \rightarrow \text{digit}$
(10)	+4\$	E*E	(23) -5	$E \rightarrow I$
(11)	+4\$	E	(115)	$E \rightarrow E*E$
(12)	4\$	E+	(115) -	
(13)	\$	E+4	(115) --	
(14)	\$	E+ I	(115) -4	$I \rightarrow \text{digit}$
(15)	\$	E+E	(115) -4	$E \rightarrow I$
(16)	\$	E	(119)	$E \rightarrow E+E$
(17)	-	E\$	(119) -	
(18)	-	S	-	$S \rightarrow E\$$

In first move, the parser shifts the state corresponding to the token digit whose LEXVAL is 2 onto the stack. On the second move, the parser reduce by the production $I \rightarrow \text{digit}$ and then invokes the semantic action $I.VAL = \text{LEXVAL}$. The VAL field of the stack is entry for digit to acquire the value 2. After each reduction and semantic action the top of the VAL stack contains the value of the translation associated with the left side of the reducing production.

Q.53. Write short note on the dependency graph. (R.G.P.V., June 2009)

Ans. If an attribute b at a node in a parse tree depends on an attribute c , then the semantic rule for b at that node must be evaluated after the semantic rule that defines c . The interdependencies among the inherited and synthesized attributes at the nodes in a parse tree can be depicted by a directed graph called a **dependency graph**.

The graph has a node for each attribute and an edge to the node for b from the node for c , if attribute b depends on attribute c . In more detail, the dependency graph for a given parse tree is constructed as follows -

for each node n in the parse tree do

for each attribute a of the grammar symbol at n do

construct a node in the dependency graph for a ;

for each node n in the parse tree do

for each semantic rule $b := f(c_1, c_2, \dots, c_k)$

associated with the production used at n do

for $i : -1$ to k do construct an edge from the node for c_i to the node for b ;

For example, suppose $A.a := f(X.x, Y.y)$ is a semantic rule for the production $A \rightarrow XY$. This rule defines a synthesized attribute $A.a$ that depends on the attributes $X.x$ and $Y.y$. If this production is used in the parse tree, then there will be three nodes $A.a$, $X.x$ and $Y.y$ in the dependency graph with an edge to $A.a$ from $X.x$ since $A.a$ depends on $X.x$ and an edge to $A.a$ from $Y.y$ since $A.a$ also depends on $Y.y$.

If the production $A \rightarrow XY$ has the semantic rule $X.i := g(A.a, Y.y)$ associated with it, then there will be an edge to $X.i$ from $A.a$ and also an edge to $X.i$ from $Y.y$. Since $X.i$ depends on both $A.a$ and $Y.y$.

Q.54. What do you understand by bottom up evaluation of S-attributed definitions ?

Or

Write short note on the S-attribute definitions. (R.G.P.V., June 2009)

Ans. Synthesized attributes can be evaluated by a bottom-up parser as the input is being parsed. The parser can keep the values of the synthesized attributes associated with the grammar symbols on its stack, whenever a

reduction is made, the values of the new synthesized attributes are computed from the attributes appearing on the stack for the grammar symbols on the right side of the reducing production.

Only synthesized attributes appear in the syntax-directed definition in fig. 2.29 for constructing the syntax tree for an expression. The approach of this can therefore be applied to construct syntax trees during bottom-up parsing. The translation of expressions during top down parsing often uses inherited attributes.

Production	Semantic Rules
$E \rightarrow E_1 + T$	$E.nptr := \text{mknode}('+', E_1.nptr, T.nptr)$
$E \rightarrow E_1 - T$	$E.nptr := \text{mknode}('-', E_1.nptr, T.nptr)$
$E \rightarrow T$	$E.nptr := T.nptr$
$E \rightarrow (E)$	$T.nptr := E.nptr$
$E \rightarrow \text{id}$	$T.nptr := \text{mkleaf}(\text{id}, \text{id.entry})$
$E \rightarrow \text{num}$	$T.nptr := \text{mkleaf}(\text{num}, \text{num.val})$

Fig. 2.29 Syntax-directed Definition for Constructing a Syntax Tree for an Expression

Q.55. Write short note on the L-attributed definitions.

(R.G.P.V., June 2009)

Ans. The order of evaluation of attributes is linked to the order in which nodes of a parse tree are created by the parsing method, when translation takes place during parsing. A new class of syntax directed definitions called L-attributed definitions, whose attributes can always be evaluated in depth-first order. Here L stands for "Left" because the information flow is left to right.

A syntax directed definition is L-attributed if each inherited attribute of x_j , $1 \leq j \leq n$, on the right side of $A \rightarrow x_1, x_2, \dots, x_n$ depends on –

- The attributes of the symbols x_1, x_2, \dots, x_{j-1} to the left of x_j in the production.
- The inherited attributes of A.
- All S-attributed definition is L-attributed.

The syntax directed definition in fig. 2.30 is not L-attributed because the inherited attribute Q.i of the grammar symbol Q depends on the attribute R.s of the grammar symbol to its right.

A translation scheme is a context free grammar in which attributes are associated with the grammar symbols and semantic actions enclosed between braces { } are inserted within the right sides of production. For example, A simple translation scheme that maps infix expressions with addition and subtraction into corresponding postfix expression.

Production	Semantic Rules
$A \rightarrow L M$	$L.i := l(A.i)$ $M.i := m(L.s)$ $A.s := f(M.s)$
$A \rightarrow Q R$	$R.i := r(A.i)$ $Q.i := q(R.s)$ $A.s := f(Q.s)$

Fig. 2.30 A Non-L-attributed Syntax-directed Definition

$E \rightarrow TR$

$R \rightarrow \text{addop } T \{ \text{print}(\text{addop.lexeme}) \} R_1 | \epsilon$

$T \rightarrow \text{num} \{ \text{print}(\text{num.val}) \}$

Fig. 2.31 shows the parse tree for the input $9 - 5 + 2$ with each semantic action attached as the appropriate child of the node corresponding to the left side of their production. Actions are terminal symbol when performed in depth first search, the action print the output $95 - 2+$. When designing a translation scheme, some restrictions are observed that an attribute value is available when an action refers to it. The easiest case occurs when only synthesized attribute are needed. The translation scheme is constructed by creating an action consisting of an assignment for each semantic rule and placing this action at the end of the right side of the associated production for example, the production and semantic rule

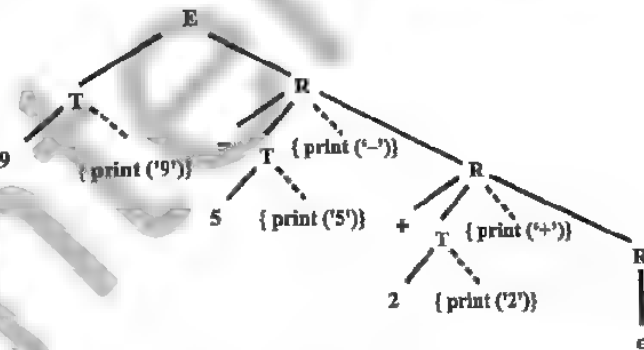


Fig. 2.31 Parse Tree for $9 - 5 + 2$ Showing Actions

Production

$T \rightarrow T_1 * F$

Semantic Rule

$T.val := T_1.val * F.val$

yield the following production and semantic actions –

$T \rightarrow T_1 * F \{ T.val := T_1.val * F.val \}$

Q.56. Explain S-attribute and L-attribute. (R.G.P.V., Dec. 2012)

Ans. Refer to Q.54 and Q.55.

Q.57. What is a syntax-directed definition? Illustrate with an example. Explain S-attribute and L-attribute definition. (R.G.P.V., May 2019)

Ans. Refer to Q.44, Q.49, Q.54 and Q.55.

Q.58. What do you understand by top-down translation? (R.G.P.V., June 2005, 2006)

Or

What do you understand by TOP down translation? Explain taking any example of your choice. (R.G.P.V., Dec. 2011)

Ans. In top-down translation, L-attributed definitions will be implemented during predictive parsing. L-attributes always be evaluated in depth-first order. In top down translation we work with translation schemes rather than syntax-directed definitions. So we can be explicit about the order in which actions and attribute evaluations take place. We eliminate left-recursion from translation

scheme as follows –

The translation scheme of fig. 2.32 is transformed into the translation scheme in fig. 2.33. The new scheme produces the annotated parse tree in fig. 2.34. for the expression $9 - 5 + 2$. The arrows in the figure suggest a way of determining the value of the expression.

In fig. 2.34, the individual numbers are generated by T, and T.val takes a value from the lexical value of the number, given by attribute num.val. The $+$ in the subexpression $9 - 5$ is generated by the leftmost T, but the minus operator and 5 are generated by the R at the right child of the root. The inherited attribute R.i obtains the value 9 from T.val. The subtraction $9 - 5$ and the parsing of the result 4 down to the middle node for R are done by embedding the following action between T and R_1 in $R_1 \rightarrow - TR_1$.

$\{ R_1.i := R.i - T.val \}$

A similar action adds 2 to the value of $9 - 5$, yielding the result $R_i = 6$ at the bottom node for R. The result is needed at the root as the value of E.val.

$E \rightarrow E_1 + T \{ E.val := E_1.val + T.val \}$

$E \rightarrow E_1 - T \{ E.val := E_1.val - T.val \}$

$E \rightarrow T \{ E.val := T.val \}$

$T \rightarrow (E) \{ T.val := E.val \}$

$T \rightarrow \text{num} \{ T.val := \text{num.val} \}$

Fig. 2.32 Translation Scheme with Left Recursion Grammar

$E \rightarrow T \{ R.i := T.val \}$

$R \{ E.val := R.s \}$

$R \rightarrow +$

$T \{ R_1.i := R.i + T.val \}$

$R_1 \{ R.s := R_1.s \}$

$R \rightarrow -$

$T \{ R_1.i := R.i - T.val \}$

$R_1 \{ R.s := R_1.s \}$

$R \rightarrow \epsilon \{ R.s := R.i \}$

$T \rightarrow ($

E

$) \{ T.val := E.val \}$

Fig. 2.33 Transformed Translation Scheme with Right-recursive Grammar

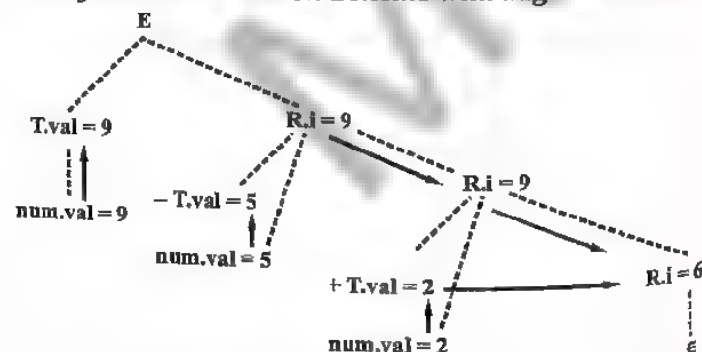


Fig. 2.34 Evaluation of the Expression $9 - 5 + 2$

Construction of a Predictive Syntax-directed Translator –

Input – A syntax-directed translation scheme with an underlying grammar suitable for predictive parsing.

Output – Code for a syntax-directed translator.

Method – The technique is a modification of the predictive-parser construction.

(i) For each nonterminal A, construct a function that has a formal parameter for each inherited attribute of A and that returns the values of the synthesized attributes of A. For simplicity, we assume that each nonterminal has just one synthesized attribute. The function for A has a local variable for each attribute of each grammar symbol that appears in a production for A.

(ii) The code for nonterminal A decides what production to use based on the current input symbol.

(iii) We consider the tokens, nonterminals, and actions on the right side of the production form left to right. The code associated with each production does the following –

(a) For token X with synthesized attribute x, save the value of x in the variable declared for X.x. Then generate a call to match token X and advance the input.

(b) For nonterminal B, generate an assignment $c := B(b_1, b_2, \dots, b_k)$ with a function call on the right side, where b_1, b_2, \dots, b_k are the variables for the synthesized attribute of B.

(c) For an action, copy the code into the parser, replacing each reference to an attribute by the variable for that attribute.

Q.59. Explain bottom-up evaluation of inherited attributes.

Ans. We give a method to implement L-attributed definitions in the framework of bottom-up parsing. This method is capable of handling all L-attributed definitions in that it can implement any L-attributed definition based on an LL(1) grammar. It can also implement many (but not all) L-attributed definitions based on LR(1) grammars. This method is a generalization of the bottom-up translation technique. The bottom-up evaluation of inherited attributes can be done as follows –

(i) By Removing Embedding Actions from Translation Scheme –

To show how inherited attributes can be handled bottom-up, we introduce a transformation that makes all embedded actions in a translation scheme occur at the right ends of their productions.

The transformation inserts new marker nonterminals generating ϵ into the base grammar. We replace each embedded action by a distinct marker nonterminal M and attach the action to the end of the production $M \rightarrow \epsilon$. For example, the translation scheme

$$\begin{aligned} E &\rightarrow TR \\ R &\rightarrow + T\{\text{print}(' + ')\} R \mid - T\{\text{print}(' - ')\} R \mid \epsilon \\ T &\rightarrow \text{num} \{\text{print}(\text{num.val})\} \end{aligned}$$

is transformed using marker nonterminals M and N into

$$\begin{aligned} E &\rightarrow TR \\ R &\rightarrow +TMR \mid -TNR \mid \epsilon \\ T &\rightarrow \text{num} \{\text{print}(\text{num.val})\} \\ M &\rightarrow \epsilon \{\text{print}(' + ')\} \\ N &\rightarrow \epsilon \{\text{print}(' - ')\} \end{aligned}$$

The grammars in the two translation schemes accept exactly the same language and, by drawing a parse tree with extra nodes for the actions, we can show that the actions are performed in the same order. Actions in the transformed translation scheme terminate productions, so they can be performed just before the right side is reduced during bottom-up parsing.

(ii) *By Inheriting Attributes on the Parser Stack* – A bottom-up parser reduces the right side of production $A \rightarrow XY$ by removing X and Y from the top of the parser stack and replacing them by A . Suppose X has synthesized attribute $X.s$, which is the implementation of bottom-up evaluation of S -attributed definitions kept along with X on the parser stack.

Since the value of $X.s$ is already on the parser stack before any reduction takes place in the subtree below Y , this value can be inherited by Y . That is, inherited attribute $Y.i$ is defined by the copy rule $Y.i := X.s$, then the value $X.s$ can be used where $Y.i$ is called for. As we shall see, copy rules play an important role in the evaluation of inherited attributes during bottom-up parsing.

(iii) *By Simulating the Evaluation of Inherited Attributes* – Reaching into the parser stack for an attribute value works only if the grammar allows the position of the attribute value to be predicted.

(iv) *By Replacing Inherited by Synthesized Attributes* – Sometimes it is possible to avoid the use of inherited attributes by changing the underlying grammar. For example, a declaration in Pascal can consist of a list of identifiers followed by a type, e.g., $m, n : \text{integer}$. A grammar for such a declaration must include productions of the form

$$\begin{aligned} D &\rightarrow L:T \\ T &\rightarrow \text{integer} \mid \text{char} \\ L &\rightarrow L, \text{id} \mid \text{id} \end{aligned}$$

Since identifiers are generated by L but the type is not in the subtree for L . We cannot associate the type with an identifier using synthesized attributes alone. In fact, if nonterminal L inherits a type from T to its right in the first production, we get a syntax-directed definition that is not L -attributed, so translations based on it cannot be done during parsing.

A solution of this problem is to restructure the grammar to include the type as the last element of the list of identifiers –

$$\begin{aligned} D &\rightarrow \text{id } L \\ L &\rightarrow , \text{id } L \mid T \\ T &\rightarrow \text{integer} \mid \text{char} \end{aligned}$$

Now, the type can be carried along as a synthesized attribute $L.type$. As each identifier is generated by L , its type can be entered into the symbol table.

Q.60. What do you understand by recursive evaluation of syntax-directed definition?

Ans. Recursive functions that evaluate attributes as they traverse a parse tree can be constructed from a syntax-directed definition using a generalization of the techniques for predictive translation. Such functions allow us to implement syntax-directed definitions that cannot be implemented simultaneously with parsing. We associate a single translation function with each nonterminal. The function visits the children of a node for the nonterminal in some order determined by the production at the node; it is not necessary that the children be visited in a left-to-right order. The traversals in recursive evaluations are as follows –

(i) *Left-to-right Traversal* – All L -attributed syntax-directed definitions can be implemented if a similar recursive function is invoked on a node for that nonterminal in a previously constructed parse tree. By looking at the production at the node, the function can determine what its children are. The function for a non-terminal A takes a node and values of the inherited attributes for A as arguments, and returns the values of the synthesized attributes for A as results.

The details of the construction are exactly as in algorithm given in Q.58, except for step (ii) where the functions for a nonterminal decide what production to use based on the current input symbol. The function here employs a case statement to determine the production used at a node.

(ii) *Other Traversals* – Once an explicit parse tree is available, we are free to visit the children of a node in any order. Consider the non- L -attributed definition of the example given below. In a translation specified by this definition, the children of a node for one production need to be visited from left to right, while the children of a node for the other production need to be visited from right to left.

This abstract example shows the power of using mutually recursive functions for evaluating the attributes at the nodes of a parse tree. The functions need not depend on the order in which the parse tree nodes are created. The main consideration for evaluation during a traversal is that the inherited attributes at a node be computed before the node is first visited and that the synthesized attributes be computed before we leave the node for the last time.

Example – Each of the nonterminals in fig. 2.35 has an inherited attribute i and a synthesized attribute s . The dependency graphs for the two productions

are also shown. The rules associated with $A \rightarrow LM$ setup left-to-right dependencies and the rules associated with $A \rightarrow QR$ setup right-to-left dependencies.

Production	Semantic Rules
$A \rightarrow LM$	$L.i := f(A.i)$ $M.i := m(L.s)$ $A.s := f(M.s)$
$A \rightarrow QR$	$R.i := r(A.i)$ $Q.i := q(R.s)$ $A.s := f(Q.s)$



Fig. 2.35 Productions and Semantic Rules for Nonterminal A

The function for nonterminal A is shown in fig. 2.36, we assume the functions for L, M, Q and R can be constructed. Variables in fig. 2.36 are named after the nonterminal and its attribute, e.g., li and ls are variables corresponding to $L.i$ and $L.s$.

```

function A(n, ai);
begin
  case production at node n of
    'A → LM': /*left-to-right order*/
      li := l(ai);
      ls := L(child(n, 1), li);
      mi := m(ls);
      ms := M(child(n, 2), mi);
      return f(ms);
    'A → QR': /*right-to-left order*/
      ri := r(ai);
      rs := R(child(n, 2), ri);
      qi := q(rs);
      qs := Q(child(n, 1), qi);
      return f(qs);
    default:
      error
  end
end;
end;

```

Fig. 2.36 Dependencies in Fig. 2.35 Determine the Order Children are Visited in

Q.61. Explain analysis of syntax-directed definition.

Ans. Attributes were evaluated during a traversal of a tree using a set of mutually recursive functions. The function for a nonterminal mapped the values of the inherited attributes at a node to the values of the synthesized attributes at that node.

This approach extends to translations that cannot be performed during a single depth-first traversal. Here we shall use a separate function for each synthesized attribute of each nonterminal, although groups of synthesized attributes can be evaluated by a single function. Earlier the construction dealt with the special case in which all synthesized attributes form one group. The grouping of attributes is determined from the dependencies setup by the

semantic rules in a syntax-directed definition. The following abstract example shows the construction of a recursive evaluation.

Example – The syntax-directed definition in fig. 2.37 is motivated by a problem we shall consider in type checking. Briefly, the problem is as follows – An “overloaded” identifier can have a set of possible types; as a result, an expression can have a set of possible types. Context information is used to select one of the possible types for each subexpression. The problem can be solved by making a bottom-up pass to synthesize the set of possible types, followed by a top-down pass to narrow down the set to a single type.

The semantic rules in fig. 2.37 are an abstraction of this problem. Synthesized attribute s represents the set of possible types and inherited attributes i represents the context information. An additional synthesized attribute t that cannot be evaluated in the same pass as s might represent the generated code or the type selected for a subexpression. Dependency graphs for the productions in fig. 2.37 are shown in fig. 2.38.

Production	Semantic Rules
$S \rightarrow E$	$E.i := g(E.s)$ $S.r := E.t$
$S \rightarrow E_1 E_2$	$E.s := fs(E_1.s, E_2.s)$ $E_1.i := fi1(E.i)$ $E_2.i := fi2(E.i)$ $E.t := ft(E_1.t, E_2.t)$
$E \rightarrow id$	$E.s := id.s$ $E.t := h(E.i)$

Fig. 2.37 Synthesized Attributes s and t cannot be Evaluated Together

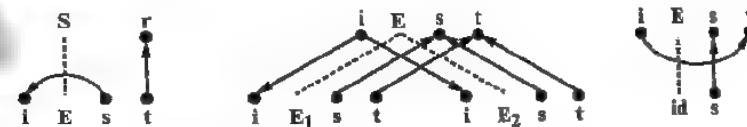


Fig. 2.38 Dependency Graphs for Productions in Fig. 2.37

Q.62. Write down the procedure to translate an infix expression into postfix form? Also write syntax directed definition for the same. (R.G.P.V., Nov. 2018)

```

Ans. #include <ctype.h>
int lookahead;
main()
{
    lookahead = getchar();
    expr();
    putchar ('\n');
}
expr()
{
    term();
    while (1)
        if(lookahead == '-')

```

```

    {
        match('-');
        term( );
        putchar('-');
    }
    elseif(lookahead == '+')
    {
        match('+');
        term( );
        putchar('+');
    }
    else break;
}
term( )
{
    if(isdigit(lookahead))
    {
        putchar(lookahead);
        match(lookahead);
    }
    else error( );
}
match(t)
int t;
{
    if(lookahead == t)
        lookahead = getchar( );
    else error( );
}
error( )
{
    printf("syntax error\n");
    exit(1); }

```

A syntax directed definition for translating expressions consisting of digits separated by plus or minus signs into postfix notation is given below –

$\text{expr} \rightarrow \text{expr}_1 + \text{term}$	$\text{expr.t} = \text{expr}_1.\text{t} \parallel \text{term.t} \parallel '+'$
$\text{expr} \rightarrow \text{expr}_1 - \text{term}$	$\text{expr.t} = \text{expr}_1.\text{t} \parallel \text{term.t} \parallel '-'$
$\text{expr} \rightarrow \text{term}$	$\text{expr.t} = \text{term.t}$
$\text{term} \rightarrow 0$	$\text{term.t} = '0'$
$\text{term} \rightarrow 1$	$\text{term.t} = '1'$
.....
$\text{term} \rightarrow 9$	$\text{term.t} = '9'$

UNIT

3

TYPE CHECKING AND RUN-TIME ENVIRONMENT

TYPE CHECKING – TYPE SYSTEM, SPECIFICATIONS OF SIMPLE TYPE CHECKER, EQUIVALENCE OF TYPE EXPRESSIONS, TYPE CONVERSION, OVERLOADING OF FUNCTIONS AND OPERATIONS, POLYMORPHIC FUNCTIONS

Q.1. Write a short note on type checking.

(R.G.P.V., June 2005, 2006, Dec. 2016)

Ans. A compiler must check the source program follows both the syntactic and semantic conventions of the source language. This checking, called *static checking*, ensures that certain kinds of programming errors will be detected and reported. Examples of static checks include –

(i) **Type Checks** – A compiler should report an error if an operator is applied to an incompatible operand; for example, if an array variable and a function variable are added together.

(ii) **Flow-of-control Checks** – Statements that cause flow of control to leave a construct must have some place to which to transfer the flow of control. For example, a break statement in C causes control to leave the smallest enclosing while, for, or switch statement, an error occurs. If such an enclosing statement does not exist.

(iii) **Uniqueness Checks** – There are situations in which an object must be defined exactly once. For example, in Pascal, an identifier must be declared uniquely, labels in a case statement must be distinct, and elements in a scalar type may not be repeated.

(iv) **Name-related Checks** – Sometimes, the same name must appear two or more times. For example, in AOA, a loop or block may have a name that appears at the beginning and end of the construct. The compiler must check that the same name is used at both places.

As the above examples indicate, most of the other static checks are routine and can be implemented using the techniques of the syntax-directed translation.

Some of them can be folded into some activities. For example, as we enter information about a name into a symbol table, we can check that the name is declared uniquely. Many Pascal compilers combine static checking and intermediate code generation with parsing.

Q.2. What do you understand by type systems? Explain.

Ans. A type system is a collection of rules for assigning type expressions to the various parts of a program. A type checker implements a type system. The type systems in type checking are specified in a syntax directed manner, so they can be readily implemented using the techniques of syntax-directed translation. Different type systems may be used by different compilers or processors of the same language. For example, in Pascal, the type of an array includes the index set of the array, so a function with an array argument can only be applied to arrays with that index set. Many Pascal compilers, however, allow the index set to be left unspecified when an array is passed as an argument. Thus these compilers use a different type system than that in the Pascal language definition.

The design of a type checker for a language is based on information about the syntactic constructs in the language, the notion of types, and the rules for assigning types to language constructs. The following excerpts from the Pascal report and the C reference manual, respectively, are examples of information that a compiler writer might have to start with.

(i) If both operands of the arithmetic operators of addition, subtraction and multiplication are of type integer, then the result is of type integer.

(ii) The result of the unary and operator is a pointer to the object referred to by the operand. If the type of the operand is '....', the type of the result is 'pointer to'.

Implicit in the above excerpts is the idea that each expression has a type associated with it. Furthermore, types have structure, the type "pointer to" is constructed from the type that "...." refers to.

In both Pascal and C, types are either basic or constructed. Basic types are the atomic types with no internal structure as far as the programmer is concerned. In Pascal, the basic types are Boolean, character, integer, and real. Subrange types, like 1.....10, and enumerated types, like

(Violet, indigo, blue, green, yellow, orange, red)

can be treated as basic types. Pascal allows a programmer to construct types from basic types and other constructed types, with arrays, records, and sets being examples.

Q.3. Explain the specification of simple type checker.

(R.G.P.V., Dec. 2014)

Ans. A type checker for a simple language is defined as in which the type of each identifier must be declared before the identifier is used. The type checker

is a translation scheme that synthesized the type of each expression from the types of its subexpression. The type checker can handle arrays, pointers, statements and functions. Position of a type checker is shown in fig. 3.1.



Fig. 3.1 Position of Type Checker

The grammar represented by the nonterminal P, consisting of a sequence of declarations D followed by a single expression E.

$$\begin{aligned}
 P &\rightarrow D; E \\
 D &\rightarrow D; D \mid \text{id}; T \\
 T &\rightarrow \text{char} \mid \text{integer} \mid \text{array} [\text{num}] \text{ of } T \mid \uparrow T \\
 E &\rightarrow \text{literal} \mid \text{num} \mid \text{id} \mid E \text{ mod } E \mid E[E] \mid E \uparrow
 \end{aligned}$$

Fig. 3.2 Grammar for Source Language

The language has two basic types – *char* and *integer*, a third basic type *type_error* is used to signal errors. For example,

array[256] of char

leads to the type expression $\text{array}(1..256, \text{char})$ consisting of the constructor array applied to the subrange 1...256 and the type char. In the translation scheme of fig. 3.3, the action associated with the production $D \rightarrow \text{id}; T$ saves a type in a symbol-table entry for an identifier. The action $\text{addtype}(\text{id.entry}, T.\text{type})$ is applied to synthesized attribute entry pointing to the symbol table entry for id and a type expression represented by synthesized attribute type of nonterminal T. If T generates char or integer, then T.type is defined to be char or integer. The upper bound of an array is obtained from the attribute val of token num that gives the integer represented by num the type constructor array is applied to the subrange 1..num.val and the element type.

Since D appears before E on the right side of $P \rightarrow D; E$, the types of all declared identifiers will be saved before the expression generated by E is checked.

$$\begin{aligned}
 P &\rightarrow D; E \\
 D &\rightarrow D; D \\
 D &\rightarrow \text{id}; T && \{ \text{addtype}(\text{id.entry}, T.\text{type}) \} \\
 T &\rightarrow \text{char} && \{ T.\text{type} := \text{char} \} \\
 T &\rightarrow \text{integer} && \{ T.\text{type} := \text{integer} \} \\
 T &\rightarrow \uparrow T_1 && \{ T.\text{type} := \text{pointer}(T_1.\text{type}) \} \\
 T &\rightarrow \text{array} [\text{num}] \text{ of } T_1 \{ T.\text{type} := \text{array}(1..\text{num.val}, T_1.\text{type}) \}
 \end{aligned}$$

Fig. 3.3 The Part of a Translation Scheme that Saves the Type of an Identifier

Q.4. What are the type expressions? Explain briefly.

Ans. The type of a language construct will be denoted by a "type expression". Informally, a type expression is either a basic type or is formed by applying an operator called a **type constructor** to other type expressions. The sets of basic types and constructors depend on the language to be checked.

The definitions of type expressions are as follows –

(i) A basic type is a type expression. Among the basic types are *Boolean*, *char*, *integer*, and *real*. A special basic type, *type_error*, will signal an error during type checking. Finally a basic type *void* denoting "the absence of a value" allows statements to be checked.

(ii) Since type expressions may be named, a type name is a type expression.

(iii) A type constructor applied to type expressions is a type expression. Constructors include –

(a) **Arrays** – If T is a type expression then $\text{array}(I, T)$ is a type expression denoting the type of an array with elements of type T and index set I . I is often a range of integers. For example, the Pascal declaration,

```
var A : array [1.....10] of integers;
```

associates the type expression $\text{array}(1.....10, \text{integer})$ with A .

(b) **Products** – If T_1 and T_2 are type expressions, then their Cartesian product $T_1 \times T_2$ is a type expression. We assume that \times associates to the left.

(c) **Records** – The difference between a record and a product is that the fields of a record have names. The *record* type constructor will be applied to a tuple formed from field names and field types. For example, the Pascal program fragment

```
type row = record
    address : integer;
    lexeme : array [1 ..... 15] of char
```

```
end;
```

```
var table : array [1 ..... 101] of row;
```

declares the type name *row* representing the type expression.

```
record(address  $\times$  integer)  $\times$  (lexeme  $\times$  array (1.....15 char))
```

and the variable *table* to be an array of records of this type.

(d) **Pointers** – If T is a type expression, then $\text{pointer}(T)$ is a type expression denoting the type "pointer to an object of type T ". For example, in Pascal, the declaration

```
var p :  $\uparrow$  row
```

declares variable p to have type *pointer (row)*.

(e) **Functions** – Mathematically, a function maps elements of one set/the *domain*, to another set/the *range*. We may treat functions in

programming languages as mapping a *domain type* D to a *range type* R . The type of such a function will be denoted by the type expression $D \rightarrow R$. For example, the built-in function *mod* of Pascal has domain type $\text{int} \times \text{int}$, i.e., a pair of integers, and range type *int*. Thus we say *mod* has the type

$$\text{int} \times \text{int} \rightarrow \text{int}$$

As another example, the Pascal declaration

```
function f(a, b : char) :  $\uparrow$  integer; .....
```

says that the domain type of f is denoted by $\text{char} \times \text{char}$ and range type by *pointer (integer)*. The type of f is thus denoted by the type expression,

$$\text{char} \times \text{char} \rightarrow \text{pointer}(\text{integer})$$

(iv) Type expressions may contain variables whose values are type expressions.

A convenient way to represent a type expressions is to use a graph. Using the syntax-directed approach, we can construct a tree or a Dag for a type expression, with interior nodes for type constructors and leaves for basic types, type names, and type variables. (see fig. 3.4).

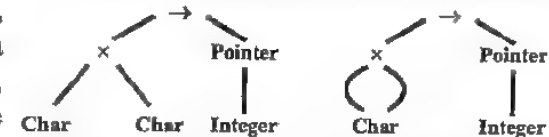


Fig. 3.4 Tree and Dag, Respectively, for $\text{Char} \times \text{Char} \rightarrow \text{Pointer}(\text{Integer})$

Q.5. Briefly discuss the structural equivalence of type expressions.

Ans. As long as type expressions are built from basic types and constructors, a natural notion of equivalence between two type expressions is *structural equivalence*, i.e., two expressions are either the same basic type, or are formed by applying the same constructor to structurally equivalent types. That is two type expressions are structurally equivalent if and only if they are identical. For example, the type expression, *integer* is equivalent only to *integer* because they are the same basic type. Similarly, *pointer(integer)* is equivalent only to *pointer(integer)* because the two are formed by applying the same constructor *pointer* to equivalent types.

```
function sequiv(s, t) : boolean;
begin
    if s and t are the same basic type then
        return true
    else if s = array(s1, s2) and t = array(t1, t2) then
        return sequiv(s1, t1) and sequiv(s2, t2)
    else if s = s1  $\times$  s2 and t = t1  $\times$  t2 then
        return sequiv(s1, t1) and sequiv(s2, t2)
    else if s = pointer(s1) and t = pointer(t1) then
        return sequiv(s1, t1)
    else if s = s1  $\rightarrow$  s2 and t = t1  $\rightarrow$  t2 then
        return sequiv(s1, t1) and sequiv(s2, t2)
    else
        return false
end
```

Fig. 3.5 Testing the Structural Equivalence of Two Type Expressions s and t

Modifications of the notion of structural equivalence are often needed in practice to reflect the actual type-checking rules of the source language. For example, when arrays are passed as parameters, we may not wish to include the array bounds as part of the type.

The algorithm for testing structural equivalence in fig. 3.5 can be adapted to test modified notions of equivalence. It assumes that the only type constructors for arrays, products, pointers and functions. The algorithm recursively compares the structure of type expressions without checking for cycles so it can be applied to a tree or a Dag representation. Identical type expressions do not need to be represented by the same node in the Dag.

The array bounds in s_1 and t_1 in

$$s = \text{array}(s_1, s_2)$$

$$t = \text{array}(t_1, t_2)$$

are ignored if the test for array equivalence in lines 4 and 5 of fig. 3.5 is reformulated as

else if $s = \text{array}(s_1, s_2)$ and $t = \text{array}(t_1, t_2)$ then return $\text{sequiv}(s_2, t_2)$

In certain situations, we can find a representation for type expressions that is significantly more compact than the type graph notation.

Q.6. Discuss the importance of type equivalence checking.

(R.G.P.V., Dec. 2012)

Ans. Refer to Q.5.

Q.7. How can we give names for type expressions?

Ans. In some languages, types can be given names. For example, in the Pascal program fragment

```
type link = ↑ cell;
```

```
var next : link;
```

```
ast : link;
```

```
p : ↑ cell;
```

```
q, r : ↑ cell;
```

the identifier *link* is declared to be a name for the type $\uparrow \text{cell}$. The question arises, do the variables *next*, *last*, *p*, *q*, *r* all have identical types? The answer of this question depends on the implementation. The problem arose because the Pascal report did not define the term "identical type".

To model this situation, we allow type expressions to be named and allow these names to appear in type expressions where we previously had only basic types. For example, if *cell* is the name of a type expression, then *pointer (cell)* is a type expression. For the time being, suppose there are no circular type expression definitions such as defining *cell* to be the name of a type expression containing *cell*.

When names are allowed in type expressions two notions of equivalence of type expressions arise, depending on the treatment of names. *Name equivalence* views each type name as a distinct type, so two type expressions are name equivalent if and only if they are identical. Under *structural equivalence*, names are replaced by the type expressions they define, so two type expressions are structurally equivalent if they represent two structurally equivalent type expressions when all names have been substituted out.

Example—The type expressions that might be associated with the variables in the declarations (1) are given in the table below—

Under name equivalence, the variables *next* and *last* have the same type because they have the same type, but *p* and *next* do not, since their associated type expressions are different. Under structural equivalence, all five variables have the same type because *link* is a name for the type expression *pointer (cell)*.

Variable	Type Expression
<i>next</i>	<i>link</i>
<i>last</i>	<i>link</i>
<i>p</i>	<i>pointer (cell)</i>
<i>q</i>	<i>pointer (cell)</i>
<i>r</i>	<i>pointer (cell)</i>

Q.8. Explain static and dynamic type checking with example.

(R.G.P.V., May 2018)

Ans. Checking done by a compiler is said to be static, while checking done when the target program runs is termed dynamic. In principle, any check can be done dynamically, if the target code carries the type of an element along with the value of that element.

A sound type system eliminates the need for dynamic checking for type errors because it allows us to determine statically that these errors cannot occur when the target program runs. That is, if a sound type system assigns a type other than *type-error* to a program part, then type errors cannot occur, when the target code for the program part is run.

In practice, some checks can be done only dynamically. For example, if we first declare

```
table : array [0 ..... 255] of char;
```

```
i : integer
```

and then compute *table[i]*, a compiler cannot in general guarantee that during execution, the value of *i* will lie in the range 0 to 255.

Q.9. What is type conversion?

(R.G.P.V., Dec. 2013)

Or

Explain briefly type conversion.

(R.G.P.V., May 2018)

Ans. Let the expressions like $x + i$ where x is of type real and i is of type integer. Since, the representation of integers and reals is different within a computer and different machine instructions are used for operations on integers

and reals, the compiler may have to first convert one of the operands of + to ensure that both operands are of the same type when the addition takes place.

The language definition specifies what conversions are necessary. When an integer is assigned to a real or vice versa, the conversion is to the type of the left side of the assignment. In expressions the usual transformation is to convert the integer into a real number and then perform a real operation on the resulting pair of real operands. The type checker in a compiler can be used to insert these conversion operations into the intermediate representation of the source program. For example, postfix notation for $x + i$, might be

$x \ i \ \text{inttooreal} \ \text{real} \ +$

Here, the **inttooreal** operator converts i from integer to real and then **real** + performs real addition on its operands.

Type conversion often arises in another context. A symbol having different meanings depending on its context is said to be overloaded.

Q.10. How type checking and type conversion is implemented in compiler ? (R.G.P.V., Dec. 2014)

Or

Explain the concept of type checking and type conversion with an example. (R.G.P.V., Dec. 2017)

Ans. Refer to Q.1 and Q.9.

Q.11. What is coercions ? Explain by giving an example.

Or

Differentiate between implicit type conversion and explicit type conversion with the help of an example. (R.G.P.V., Dec. 2010, 2011)

Or

Explain implicit type conversion and explicit type conversion. (R.G.P.V., June 2010)

Or

Compare explicit and implicit type conversion. (R.G.P.V., Dec. 2012)

Ans. Conversion from one type to another is said to be *implicit* if it is to be done automatically by the compiler. Implicit type conversions also called *coercions*, are limited in many languages to situations where no information is lost in principle e.g., an integer may be converted to a real but not vice versa. In practice, however, loss is possible when a real number must fit into the same number of bits as an integer.

Conversion is said to be *explicit* if the programmer must write something to cause the conversion. For all practical purposes, all conversions in Ada are explicit. Explicit conversions look just like function applications to a type checker, so they present no new problems.

For example, in Pascal, a built-in function *ord* maps a character to an integer and *chr* does the inverse mapping from an integer to a character, so these

conversions are explicit. C, on the other hand, coerces (i.e., implicitly converts) ASCII characters to integers between 0 and 127 in arithmetic expressions.

Example – Consider expressions formed by applying an arithmetic operator **op** to constants and identifiers, as in the grammar of fig. 3.6. Suppose there are two types—real and integer, with integers converted to reals when necessary. Attribute *type* of nonterminal *E* can be either integer or real and the type checking rules are shown in fig. 3.6.

Production	Semantic Rule
$E \rightarrow \text{num}$	$E.\text{type} := \text{integer}$
$E \rightarrow \text{num.num}$	$E.\text{type} := \text{real}$
$E \rightarrow \text{id}$	$E.\text{type} := \text{lookup}(\text{id.entry})$
$E \rightarrow E_1 \text{ op } E_2$	$E.\text{type} :=$ if $E_1.\text{type} = \text{integer}$ and $E_2.\text{type} = \text{integer}$ then integer else if $E_1.\text{type} = \text{integer}$ and $E_2.\text{type} = \text{real}$ then real else if $E_1.\text{type} = \text{real}$ and $E_2.\text{type} = \text{integer}$ then real else if $E_1.\text{type} = \text{real}$ and $E_2.\text{type} = \text{real}$ then real else type_error

Fig. 3.6 Type-checking Rules for Coercion from Integer to Real

Q.12. What do you mean by overloading of functions and operations ?

(R.G.P.V., June 2016)

Or

Explain overloading of functions and operations briefly.

(R.G.P.V., Dec. 2013)

Ans. An overloaded symbol is one that has different meanings depending on its context. In mathematics, the addition operator + is overloaded, because + in $A + B$ has different meanings when A and B are integers, reals, complex numbers, or matrices. In Ada parentheses () are overloaded; the expression $A(I)$ can be the *I*th element of the array A, a call to function A with argument I, or an explicit conversion of expression I to type A.

Overloading is *resolved* when a unique meaning for an occurrence of an overloaded symbol is determined. For example, if + can denote either integer addition or real addition, then the two occurrences of + in $x + (i + j)$ can denote different forms of addition, depending on the types of x , i and j . The resolution of overloading is sometimes referred to as *operator identification*, because it determines which operation an operator symbol denotes.

The arithmetic operators are overloaded in most languages. However, overloading involving arithmetic operators like + can be resolved by looking only at the arguments of the operator.

Q.13. What is polymorphic functions ?

(R.G.P.V., Dec. 2014)

Or

Write a short note on polymorphic functions. (R.G.P.V., June 2017)

Ans. An ordinary procedure allows the statements in its body to be executed with arguments of fixed types; each time a polymorphic procedure is called, the statements in its body can be executed with arguments of different types. The term polymorphic can also be applied to any piece of code that can be executed with arguments of different types. Built-in operators for indexing arrays, applying functions, and manipulating pointers are usually polymorphic because they are not restricted to a particular type of array, function, or pointer.

Q.14. Why do we need polymorphic functions ?

Ans. Polymorphic functions are attractive because they facilitate the implementation of algorithms that manipulate data structures, regardless of the types of the elements in the data structure. For example, it is convenient to have a program that determines the length of a list without having to know the types of the elements on the list.

Languages like Pascal require full specification of the types of function parameters, so a function for determining the length of a linked list of integers cannot be applied to a list of reals. The Pascal code in fig. 3.7 is for lists of integers. The function *length* follows the next links in the list until a *nil* link is reached. Although the function does not in any way depend on the type of the information in a cell Pascal requires the type of the *info* field to be declared when the *length* function is written.

```

type link = ↑ cell;
cell = record
    info : integer;
    next : link
end;

var len : integer;
begin
    while lptr <> nil do begin
        len := len + 1;
        lptr := lptr ↑ .next
    end;
    length := len
end;

function length (lptr : link) : integer;
begin
    len := 0;
    while lptr <> nil do begin
        len := len + 1;
        lptr := lptr ↑ .next
    end;
    length := len
end;

```

Fig. 3.7 Pascal Program for the Length of a List

In a language with polymorphic functions, like ML (Milner 1984), a function *length* can be written so it applies to any kind of list, as shown in fig. 3.8. The

keyword *fun* indicates that *length* is a recursive function. The functions *null* and *t1* are predefined; *null* tests if a list is empty and *t1* returns the remainder of the list after the first element is removed. With the definition shown in fig. 3.8, both the following applications of the function *length* yield 3 -

```
length(["sun", "mon", "tue"]);
```

```
length([10, 9, 8]);
```

In the first, *length* is applied to a list of strings; in the second, it is applied to a list of integers.

```

fun length(lptr) =
  if null(lptr) then 0
  else length(t1(lptr)) + 1;

```

Fig. 3.8 ML Program for the Length of a List

Q.15. Describe the checking rules for the polymorphic functions.

(R.G.P.V., June 2012)

Ans. The rules for checking expressions generated by the grammar in fig. 3.9 will be written in terms of the following operations on a graph representation of types -

(i) *fresh(t)* replaces the bound variables in type expression *t* by fresh variables and returns a pointer to a node representing the resulting type expression. Any \forall symbols in *t* are removed in the process.

(ii) *unify(m, n)* unifies the type expressions represented by the nodes pointed to by *m* and *n*. It has the side effect of keeping track of the substitution that makes the expressions equivalent. If the expressions fail to unify, the entire type-checking process fails.

Individual leaves and interior nodes in the type graph are constructed using operations *mkleaf* and *mknnode* similar to those of construction of syntax trees. It is necessary that there be a unique leaf for each type variable, but other structurally equivalent expressions need not have unique nodes.

The *unify* operation is based on the following graph-theoretic formulation of unification and substitutions. Suppose nodes *m* and *n* of a graph represent expressions *e* and *f*, respectively. We say nodes *m* and *n* are *equivalent* under substitution *S* if $S(e) = S(f)$. Also two nodes *m* and *n* are equivalent if and only if they represent the same operator and their corresponding children are equivalent.

The type checking rules for expressions are shown in fig. 3.10. We do not show how declarations are processed. As type expressions generated by

```

P → D ; E
D → D ; D | id : Q
Q → ∀ type variable.Q | T
T → T' → T
    | T × T
    | unary_constructor (T)
    | basic_type
    | type_variable
    | (T)
E → E(E) | E, E | id

```

Fig. 3.9 Grammar for Language with Polymorphic Functions

nonterminals T and Q are examined, $mkleaf$ and $mknode$ add nodes to the type graph, following the Dag construction in construction of syntax trees. When an identifier is declared, the type in the declaration is saved in the symbol table in the form of a pointer to the node representing the type. In fig. 3.10, this pointer is referred to as the synthesized attribute $id.type$. As mentioned above the *fresh* operation removes the \forall symbols as it replaces bound variables by fresh variables. The action associated with production $E \rightarrow E_1, E_2$ sets $E.type$ to the product of the types of E_1 and E_2 .

```

 $E \rightarrow E_1(E_2)$   {  $p := mkleaf(newtypevar);$ 
                   $unify(E_1.type, mknode(' \rightarrow ', E_2.type, p));$ 
                   $E.type := p$  }
 $E \rightarrow E_1, E_2$   {  $E.type := mknode(' \times ', E_1.type, E_2.type)$  }
 $E \rightarrow id$        {  $E.type := fresh(id.type)$  }

```

Fig. 3.10 Translation Scheme for Checking Polymorphic Functions

The type checking rule for the function application $E \rightarrow E_1(E_2)$ is motivated by considering the case where $E_1.type$ and $E_2.type$ are both type variables, say $E_1.type = \alpha$ and $E_2.type = \beta$. Here $E_1.type$ must be a function such that for some unknown type γ , we have $\alpha = \beta \rightarrow \gamma$. In fig. 3.9, a fresh type variable corresponding to γ is created and $E_1.type$ is unified with $E_2.type \rightarrow \gamma$. A new type variable is returned by each call of *newtypevar*, a leaf for it is constructed by *mkleaf* and a node representing the function to be unified with $E_1.type$ is constructed by *mknode*. After unification succeeds, the new leaf represents the result type.

RUN-TIME ENVIRONMENT – STORAGE ORGANIZATION, STORAGE ALLOCATION STRATEGIES, PARAMETER PASSING, DYNAMIC STORAGE ALLOCATION, SYMBOL TABLE

Q.16. What is run-time environment? What are the important elements of run-time environment? How is it controlled in a program that is compiled? (R.G.P.V., Dec. 2015)

Ans. Run-time environment is a term used to broadly describe all the runtime settings of a program in execution. The important elements of run-time environment are as follows –

(i) **Memory Organization** – The program needs memory for storing local variables, global variables, the code of the program and so on at the time of execution. Memory organization during execution is an important aspect of run-time environment. Run-time memory organization is determined by the features of the source language.

(ii) **Activation Records** – For making programs modular, source languages support procedure or functions. Procedure activation is managed by having a contiguous block of memory known as activation record. The activation record contains among other things, the memory for all the local variables of the procedure. A single activation record which is common across any number of activations can be created statically. The activation record can also be constructed dynamically, one for each activation. The target code has to be generated accordingly to access the local variables that are part of it depending on how the activation record is created.

(iii) **Procedure Calling and Return Sequences** – There are certain sequence of operations when a procedure is activated. This sequence of operations which is carried out during the process of calling a procedure is known as the calling sequence. Similarly, there would be a sequence of actions to be performed when an activation procedure completes execution. This sequence of actions to be carried out when a procedure returns after completing its execution is known as return sequence.

(iv) **Parameter Passing** One or more parameters can be passed when a function is called. The called function might modify the value of the parameter. There exist several mechanisms by which parameters can be passed to functions. The target code generator should take into account the type of the parameter passing mechanism used in the context and generate code accordingly.

The runtime environment is indirectly controlled by generating the code to maintain it in case of compiled languages. In case of an interpreted program, the run-time environment is maintained directly in the data structures of the interpreter.

Q.17. How can the run-time memory be subdivided? Explain in brief.

Ans. Suppose that the compiler obtains a block of storage from the operating system for the compiled program to run in. This run-time storage might be sub-divided to hold –

- The generated target code
- Data objects, and
- A counter part of the control stack to keep track of procedure activations.

The size of the generated target code is fixed at compile time, so the compiler can place it in a statically determined area, perhaps in the low end of memory. Similarly, the size of the some data objects may also be known at compile time, and these too can be placed in a statically determined area, as in fig. 3.11. One reason for statically allocating as many data objects as possible is that the addresses of these objects can be compiled into the target code. All data objects in Fortran can be allocated statically.

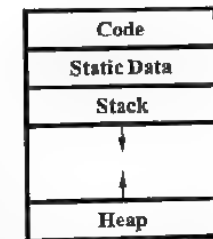


Fig. 3.11 Typical Subdivision of Run-time Memory into Code and Data Areas

Implementations of languages like Pascal and C use extensions of the control stack to manage activations of procedures. When a call occurs, the execution of an activation is interrupted and information about the status of the machine, such as the value of the program counter and machine register, is saved on the stack. When control returns from the call, this activation can be restarted after restoring the values of relevant registers and setting the program counter to the point immediately after the call. Data objects whose lifetimes are contained in that of an activation can be allocated on the stack along with other information associated with the activation.

A separate area of run-time memory, called a *heap*, holds all other information. Pascal allows data to be allocated under program control; the storage for such data is taken from the heap. Implementations of languages in which the lifetimes of activations cannot be represented by an activation tree might use the heap to keep information about activations. The controlled way in which data is allocated and deallocated on a stack makes it cheaper to place data on the stack than on the heap.

The sizes of the stack and the heap can change as the program executes, so we show these at opposite ends of memory in fig. 3.11, where they can grow toward each other as needed. Pascal and C need both a run-time stack and heap, but not all languages do.

Q.18. Explain the importance of run-time storage management in compilers. (R.G.P.V., June 2012)

Ans. Refer to Q.17.

Q.19. Explain the memory allocation in block structured languages. (R.G.P.V., Dec. 2013)

Ans. A block is a program unit which can contain data declarations. A program in a block structured language is a nested structure of blocks. A block structured language uses dynamic memory allocation. Automatic dynamic allocation is implemented using the extended stack model with a minor variation – each record in the stack has two reserved pointers instead of one as shown in fig. 3.12. Each stack record accommodates the variables for one activation of a block, hence we call it an activation record (AR). The following notation is used to refer to the activation record of a block.

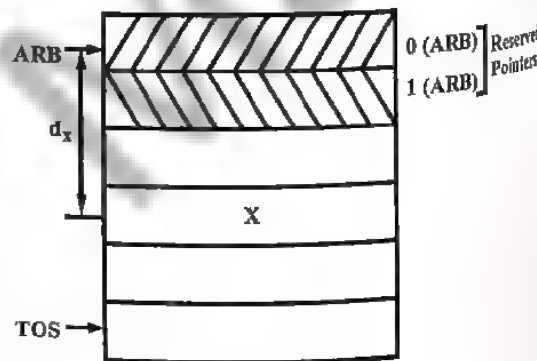


Fig. 3.12

AR_A^i – Activation record for the i^{th} activation of A wherein we omit the superscript i unless multiple activations of a block exist. A register called the activation record base (ARB) always points to the start address of the TOS record during the execution of a block structured program. This record belongs to the block which contains the statement being executed. A local variable X of this block is accessed using the address d_X (ARB), where d_X is the displacement of variable X from the start of AR. The address may also be written as $\langle ARB \rangle + d_X$, where $\langle ARB \rangle$ stands for the words 'contents of ARB'.

Q.20. What is activation record? Give the general activation record field and their purpose. (R.G.P.V., Dec. 2006, 2007, 2010, June 2012)

Or

Write a short note on activation record. (R.G.P.V., June 2008, 2011, 2016)

Or

What is activation record? Explain each of its fields.

(R.G.P.V., June 2010)

Or

What do you mean by activation record? Why this record is maintained by compiler? Explain various fields of activation record.

(R.G.P.V., Dec. 2008)

Or

What is an activation record? With the help of diagram show the important fields in an activation record. (R.G.P.V., Dec. 2015)

Ans. Information needed by a single execution of a procedure is managed using a contiguous block of storage called an *activation record* or *frame*, consisting of the collection of fields shown in fig. 3.13. Not all languages nor all compilers use all of these fields; often registers can take the place of one or more of them. For languages like Pascal and C, it is customary to push the activation record of a procedure on the run-time stack when the procedure is called and to pop the activation record off the stack when control returns to the caller.

The purpose of the fields of an activation record is as follows, starting from the field for temporaries –

(i) Temporary values, such as those arising in the evaluation of expressions, are stored in the field for temporaries.

(ii) The field for local data holds data that is local to an execution of a procedure.

(iii) The field for save machine status holds information about the state of the machine just before the procedure is called. This information includes the

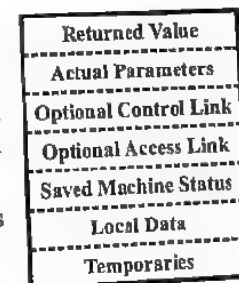


Fig. 3.13 A General Activation Record

values of the program counter and machine registers that have to be restored when control returns from the procedure.

(iv) The optional *access link* is used in access to nonlocal names to refer to nonlocal data held in other activation records. For a language like Fortran access links are not needed because nonlocal data is kept in a fixed place. Access links, or the related “display” mechanism, are needed for Pascal.

(v) The optional *control link* points to the activation record of the caller.

(vi) The field for actual parameters is used by the calling procedure to supply parameters to the called procedure. We show space for parameters in the activation record, but in practice parameters are often passed in machine registers for greater efficiency.

(vii) The field for the returned value is used by the called procedure to return a value to the calling procedure. Again, in practice this value is often returned in a register for greater efficiency.

The sizes of each of these fields can be determined at the time a procedure is called. Infact, the sizes of almost all fields can be determined at compile time. An exception occurs if a procedure may have a local array whose size is determined by the value of an actual parameter, available only when the procedure is called at run-time.

Q.21. What do you understand by compile-time layout of local data?

Ans. The field for local data is laid out as the declarations in a procedure are examined at compile time. Variable-length data is kept outside this field. We keep a count of the memory locations that have been allocated for previous declarations. From the count we determine a *relative* address of the storage for a local with respect to some position such as the beginning of the activation record. The relative address, or *offset* is the difference between the addresses of the position and the data object.

The storage layout for data objects is strongly influenced by the addressing constraints of the target machine. For example, instructions to add integers may expect integers to be *aligned*, that is, placed at certain positions in memory such as an address divisible by 4. Although an array of ten characters needs only enough bytes to hold ten characters, a compiler may therefore allocate 12 bytes, leaving 2 bytes unused. Space left unused due to alignment considerations is referred to as *padding*. When space is at a premium, a compiler may pack data so that no padding is left; additional instructions may then need to be executed at runtime to position packed data so that it can be operated on as if it were properly aligned.

Q.22. Explain various storage allocation strategies. Which storage allocation is to be used if language permits recursion?

(R.G.P.V., June 2003, 2005, 2006, 2009, Dec. 2011)

Or

Explain the difference between static, stack and heap allocation.

(R.G.P.V., June 2010)

Or

Explain various storage allocation strategies. (R.G.P.V., Dec. 2015)

Or

Explain different storage allocation strategies with the help of suitable examples. (R.G.P.V., Dec. 2016)

Or

Write different storage allocation strategies. (R.G.P.V., June 2017)

Ans. A different storage allocation strategy is used in each of the three data areas in the organization –

(i) **Static Allocation** – Static allocation lays out storage for all data object at compile time. In static allocation, names are bound to storage as the program is compiled, so there is no need for a run-time support package. The bindings do not change at run-time, every time a procedure is activated, its names are bound to the same storage locations. This property allows the values of local names to be retained across activations of a procedure. From the type of a name, the compiler determines the amount of storage to set aside for that name.

The address of this storage consists of an offset from an end of the activation record for the procedure. The compiler decides where the activation records go, relative to the target code and to one another. After this the position of each activation record and storage for each name in the record is fixed. At compile-time, the addresses are filled where the target code finds the data that it operates on the addresses at which information is also save when a procedure call occurs, are also known at compile-time.

Static allocation has following limitations –

(a) The size of a data object and constraints on its position in memory must be known at compile-time.

(b) Recursive procedures are restricted, because all activations of a procedure use the same bindings for local names.

(c) Data structures cannot be created dynamically, since there is no mechanism for storage allocation at runtime.

(ii) **Stack Allocation** – Stack allocation is based on the idea of a control stack. Storage is organized as the stack, and activation records are pushed and popped as activations begin and end, respectively. Storage for the locals in each call of a procedure is contained in the activation record for that call. Thus, locals are bound to fresh storage in each activation, because a new

activation record is pushed onto the stack when a call is made. The values of locals are deleted when the activation ends i.e., the value are lost because the storage for locals disappears when the activation record is popped.

The register *top* marks the top of the stack at run time, an activation record can be allocated and deallocated by incrementing and decrementing *top* respectively, by the size of the record. If procedure *q* has an activation record of size *a*, then *top* is incremented by *a*, just before the target code *q* is executed. When control returns from *q*, *top* is decremented by *a*. Procedure calls are implemented by generating a calling sequence in the target code. A call sequence allocates an activation record and enters information into its fields. A return sequence restores the state of the machine so the calling procedure can continue execution. The code in a calling sequence is often divided between the calling procedure and the procedure it calls.

Position in Activation Tree	Activation Records on the Stack	Remarks
		Frame for <i>s</i>
		<i>r</i> is activated
		Frame for <i>r</i> has been popped and <i>q</i> (1,9) pushed
		Control has Just Returned to <i>q</i> (1,3)

Fig. 3.14 Downward-growing Stack Allocation of Application Records

The call sequence is –

- The caller evaluates actuals.
- The caller stores a return address and the old value of *top_sp* into the callee's activation record the caller then increments *top_sp* i.e., it is moved past the caller's local data and temporaries and the callee's parameter and status fields. Here, *top_sp* is the register points to the end of the machine status field in an activation record.

- The callee saves register values and other status information.
- The callee initializes its local data and begins execution.

The return sequence is –

- The callee places a return value next to the activation record of the caller.
- The callee restores *top_sp* and other registers and branches to a return address in the caller's code according to the information in the status field.
- top_sp* is decremented, the caller can copy the returned value into its own activation record and use it to evaluate an expression.

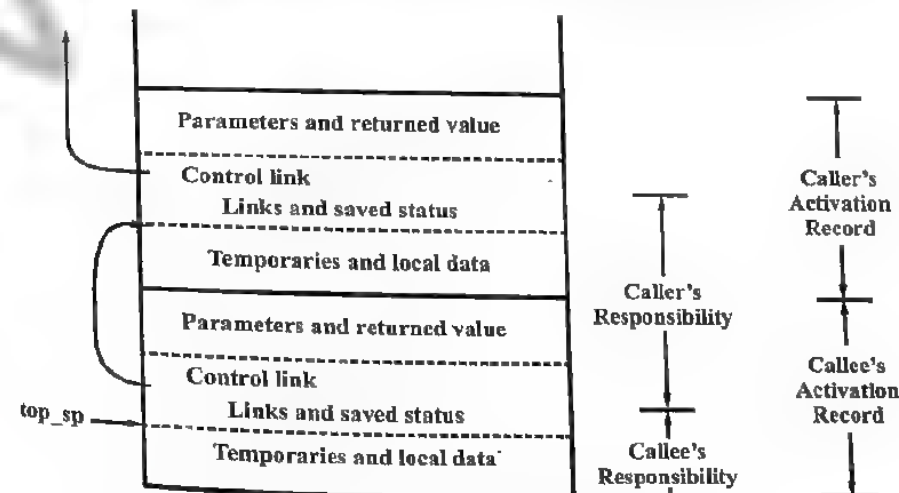


Fig. 3.15 Division of Task between Caller and Callee

A dangling reference occurs when there is a reference to storage that has to be deallocated. It is a logical error to use dangling references, since the value of deallocated storage is undefined according to the semantic of most language. Stack storage allocation is to be used if language permits recursion.

(iii) **Heap Allocation** – Heap allocation strategy is used if either of the following is possible –

(a) The values of local names must be retained when an activation ends.

(b) A called activation outlives the caller. This possibility cannot occur for those language where activation trees correctly depict the flow of control between procedures.

The storage in above cases cannot be organized as a stack, because, the deallocation of activation records need not occur in a last-in first-out.

Heap allocation parcels out pieces of contiguous storage, as needed for activation records or other objects. Pieces may be deallocated in any order. The heap allocation is shown in fig. 3.16, the record for the activation of procedure *r* is retained when the activation ends. The record for the new activation *q* (1,9) cannot follow that for *s*. If the retained activation record for *r* is deallocated, there will be free space in the heap between the activation records for *s* and *q* (1,9). It is left to the heap manager to make use of this space. There is some time and space overhead by using a heap manager. For efficiency reasons, it may be helpful to handle small activation records or records of a predictable size as follows –

(a) For each size, keep a linked list of free blocks of that size.
 (b) Fill a request for size *s* with a block of size *s'*, where *s'* is the smallest size greater than or equal to *s* when the block is deallocated, it is returned to the linked list.

(c) For large blocks of storage use the heap manager.

This approach gives fast allocation and deallocation of small amount of storage.

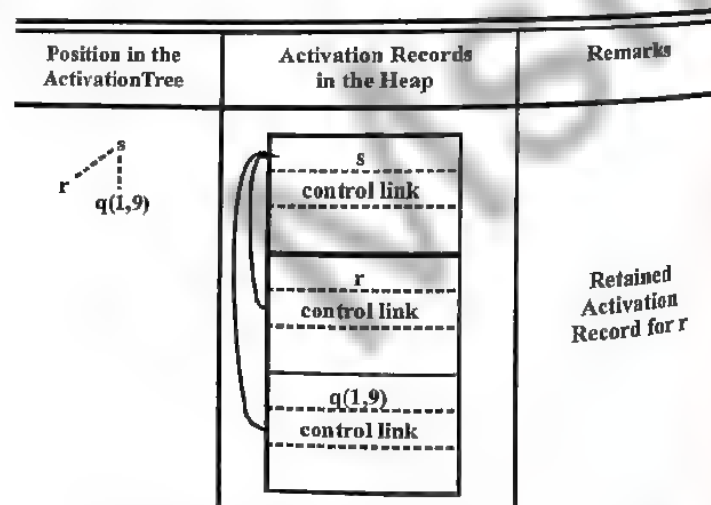


Fig. 3.16 Records for Live Activation in a Heap

Q.23. Explain activation trees and activation records with an suitable examples. Also explain calling sequence. (R.G.P.V., May 2019)

Ans. An activation tree is used to show the flow of control in a program. In an activation tree –

- Each of the function activation represents a node.
- The activation of the main program is represented by the root.
- If control flows from activation *a* to *b* then the node for *a* is the parent of the node for *b*.
- If the lifetime of *a* occurs before the lifetime of *b* then the node for *a* is to the left of the node for *b*.

```

program sorting (input, output);
var arr: array [0..10] of integer;
procedure readarray;
var i: integer;
begin
  for i := 1 to 9 do read (arr[i]);
end;
function partition (y, z: integer): integer;
var i, j, x, v: integer;
begin...
end;
procedure quicksort (m, n: integer);
var i: integer;
begin
  if (n > m) then begin
    i := partition (m, n);
    quicksort (m, i - 1);
    quicksort (i + 1, n);
  end
end;
begin
  arr[0] := - 9999; arr[10] := 9999;
  readarray;
  quicksort (1, 9)
end

```

Output –
 Execution begins. . .
 enter readarray
 leave readarray
 enter quicksort (1, 9)
 enter partition (1, 9)
 leave partition (1, 9)
 enter quicksort (1, 3)
 leave quicksort (1, 3)
 enter quicksort (5, 9)
 leave quicksort (5, 9)
 leave quicksort (1, 9)
 Execution terminated

Fig. 3.17 A Pascal Program for Reading and Sorting Integers

Fig. 3.18 shows an activation tree corresponding to the output in fig. 3.17. Only the first letter of each procedure is shown to save space. The root of the activation tree is for the entire program sort. There is an activation of *readarray* during the execution of *sort*. This is represented by the first child of the root with label *r*. The next activation is for *quicksort* with actuals 1 and 9, represented by the second child of the root. During this activation, the calls of *partition* and *quicksort* lead to the activation *p*(1, 9), *q*(1, 3), and *q*(5, 9). It is noted that the activations *q*(1, 3) and *q*(5, 9) are recursive, and that they begin and end before *q*(1, 9) ends.

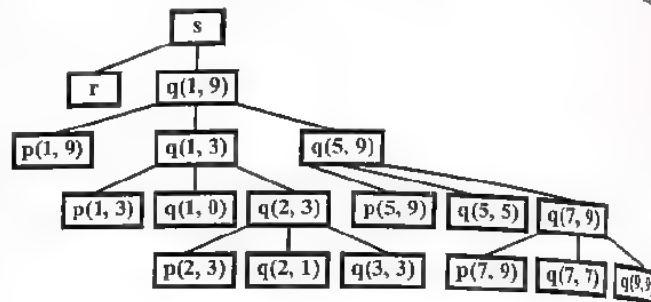


Fig. 3.18 Activation Tree

Refer to Q.20 and Q.22 (ii).

Q.24. What are the limitations of static allocation ? (R.G.P.V., June 2016)

Ans. Refer to Q.22 (i).

Q.25. Discuss the following storage-allocation strategies –

(i) **Stack allocation** (ii) **Heap allocation.**

(R.G.P.V., Dec. 2012)

Ans. Refer to Q.22 (ii) and (iii).

Q.26. Differentiate between stack allocation and heap allocation.

(R.G.P.V., June 2016, Nov. 2019)

Ans. The differences between the stack allocation and heap allocation are as follows –

S.No.	Stack Allocation	Heap Allocation
(i)	Stack allocation is useful for handling recursive procedures.	Heap allocation is useful for implementing data whose size varies as the program is running.
(ii)	In stack allocation, a region of consecutive words of memory is used.	In heap allocation, a large block of storage partitioned into smaller block is used.
(iii)	It is used in ALGOL.	It is used in SNOBOL and LISP.
(iv)	Stack allocation is used where the size of the data structure is deterministic.	Heap allocation is used, where the size of the data structure is non-deterministic.

- (v)
- | | |
|---|---|
| In stack allocation, when a procedure is called, it places its data on top of a stack and when the procedure returns, it pops its data off the stack. | In heap allocation, large block of memory is divided into variable length blocks, some used for data and some free. When a piece of data is created, we must find a free block of sufficient size. When data is no longer needed, its block is freed. |
|---|---|

Q.27. What do you mean by heap allocation ? Explain the various terms related to heap allocation –

(i) **Free list** (ii) **Reference count** (iii) **Fragmentation.**

(R.G.P.V., June 2009, Dec. 2011)

Ans. Heap Allocation – Refer to Q.22 (iii).

(i) **Fragmentation** – When we free a block, we must do something to attach it to adjacent free block, if any. The penalty for not doing so is that storage will fragment, that is the available space list will consist of many little blocks, none of which is sufficient to hold a large block of data. When no sufficiently large block of the heap is available, a library routine loaded with the program must merge adjacent free block into larger blocks. It may even be necessary to move data around in the heap to make a large consecutive free block.

(ii) **Free List** – A free list is a data structure used in a scheme for dynamic memory allocation. It operates by connecting unallocated regions of memory together in a linked list, using the first word of each unallocated region as a pointer to the next. It is most suitable for allocating from a memory pool, where all objects have the same size.

Free list makes the allocation and deallocation operations very simple. To free a region, we just add it to the free list. To allocate a region, we simply remove a single region from the end of the free list and use it. If the region are variable sized, we may have to search for a region of large enough size, which can be expensive.

There are following disadvantages of free list – Inherited from linked lists, of poor locality of reference and so poor data cache utilization, and they provide no way of consolidating adjacent regions to fulfill allocations requests for large regions, unlike the buddy allocation system.

(iii) **Reference Counts** – We keep track of the number of blocks that point directly to the present block. If this count ever drops to 0, then the block can be deallocated because it cannot be referred to. In other words, the block has become garbage that can be collected. Maintaining reference counts can be costly in time the pointer assignment $p = q$ leads to changes in the reference counts of the blocks pointed to by both p and q . The count for the block

pointed to by p goes down by one, while that for the block pointed to by q goes up by one. Reference counts are best used when pointers between blocks never appear in cycles. For example in fig. 3.19 shown, neither block is accessible from any other, so they are both garbage, but each has a reference count of one.

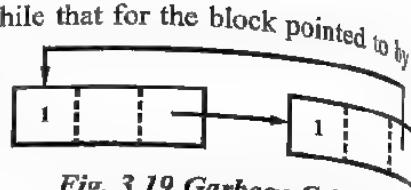


Fig. 3.19 Garbage Cells with Non-zero Reference Counts

Q.28. What is the difference between dynamic and static storage management? Explain the importance of run-time storage management in compiler.

(R.G.P.V., June 2008, 2011)

Ans. Static Storage Management – Refer to Q.22 (i).

Dynamic Storage Management – Refer to Q.22 (iii) under heap allocation.

Run-time Storage Memory Management – Refer to Q.17.

Q.29. Write short note on parameter passing.

(R.G.P.V., Dec. 2003, Dec. 2005, 2006, 2008, June 2011)

Or

Explain with a suitable example, mechanism used by a compiler to handle procedure parameters.

(R.G.P.V., June 2010)

Or

Describe parameter passing mechanism for a procedure call.

(R.G.P.V., June 2009, Dec. 2010)

Or

Explain the various parameter passing mechanism. (R.G.P.V., Dec. 2010)

Ans. integer procedure DIVIDE (X,Y) integer X,Y :

If Y = 0 then return 0

else return X/Y

Defines a procedure called DIVIDE. X and Y in this definition are called formal parameter or just formals.

A : DIVIDE (B,C)

B and C are called actual parameters, or actuals.

The formal parameters of a procedure definition are placeholders for values which will be supplied when the procedure is called. The actual parameters provide the values to be substituted for the formal parameters. The compiler can treat this situation as if there were an additional formal and actual parameter associated with the function procedure.

There are three common methods of passing parameters. They are –

(i) **Call-by-Value** – This is the simplest possible method of passing parameters. The actual parameters are evaluated and their r-values are passed to the subroutine in locations determined by the language implementation. The

effect is that of initializing the formal parameters of the subroutine with the r-values of the actual parameters. Call-by-value can be implemented as follows –

(a) A formal parameter is treated as a local name, so the storage for the formals is in the activation record of the called procedure.

(b) The caller evaluates the actual parameters and places their r-values in storage for the formals.

For example,

A distinguishing feature of call-by-value is that operations on the formal parameters do not affect values in the activation record of the caller when the control returns from the call and the activation record for swap is deallocated. The call has no effect on the activation record of the caller. The following program illustrates the 'call by value' –

```
main ( )
{
    int a = 10, b = 20;
    swap(a,b);
    printf("a = %d b = %d", a, b);
}

swap(int x, int y)
{
    int t;
    t = x;
    x = y;
    y = t;
    printf("\n x = %d y = %d", x, y );
}
```

The output of the above program would be –

x = 20, y = 10

a = 10, b = 20

(ii) **Call-by-Reference** – The address of actual arguments in the calling procedure are copied into formal arguments of the called procedure. This means that using these addresses we would have an access to the actual arguments and hence we would be able to manipulate them. The following program illustrates this fact –

```
main ( )
{
    int a = 10, b = 20;
    swap(&a, &b);
    printf("\n a = %d b = %d", a, b);
}
```

```

swap(int *x, int *y)
{
    int t;
    t = *x;
    *x = *y;
    *y = t;
}

```

In this program compiler manages to exchange the value of a and b using their addresses stored in x and y.

(iii) **Copy-Restore** – A generalization of call-by-value is copy-restore (copy-in, copy-out, or value-result) linkage. A hybrid between call by value and call by reference is copy restore linkage.

(a) Before control flows to the called procedure the actual parameters are evaluated. The r-values of the actuals are passed to the caller procedure as in call-by-value.

(b) When control returns, the current r-values of the formal parameters are copied back into the l-values of the actuals using the l-values are copied.

The first step “copies-in” the values of the actuals into the activation record of the called procedure. The second step “copies out” the final values of the formals into the activation record of the caller.

If we verify that SWAP(I, A[I]) will work correctly using copy-restore linkage. The important point is that the location A[I] is computed and preserved by the calling program before the call. Thus, the final value of formal parameter Y, which will be the initial value of I, is copied into the correct location, even though, should we compute the location of A[I] after the call, we would get a different result (because the value of I has changed).

(iv) **Call-by-Name** – Call-by-name is defined by the copy rule of Algol, which is –

(a) The procedure is treated with the actual parameters substituted for the formals if it were a macro, i.e., its body is substituted for the call in the caller. Such a literal substitution is called macro-expansion or in-line expansion.

(b) The local names of called procedure are different from the names of the calling procedure.

(c) The actual parameters are surrounded by parentheses to preserve their integrity.

The implementation of call by name is to pass to the called procedure parameterless subroutines called thunks, that can evaluate the l-value or r-value of the actual parameters. A thunk carries an access link with it, pointing to the current activation record for the calling procedure.

Q.30. Distinguish between static scope and dynamic scope. Briefly explain access to non-local names in static scope. (R.G.P.V., Dec. 2013)

Ans. The method of binding names to non-local variables, is called static scoping. Static scoping is so named because the scope of a variable can be statically determined, that is, prior to execution. Dynamic scoping is based on the calling sequence of subprograms, not on their spatial relationship to each other. Thus the scope can be determined only at run-time.

Static scope rules determine which declaration a name's reference will be associated with, depending upon the program's language, thereby determining from where the name's value will be obtained at run-time. When static scope rules are used during compilation, the compiler knows how the declarations are bound to the name references, and hence, from where their values will be obtained at run-time. When the compiler has to do is to provision the retrieval of the non-local name value when it is accessed at run-time.

Q.31. What are the language facilities for dynamic storage allocation? (R.G.P.V., June 2017)

Ans. Some languages provide facilities for the dynamic allocation of storage for data under program control. For such data, storage is usually taken from heap. Allocated data is frequently retained until it is explicitly deallocated. The allocation can be either explicit or implicit. In Pascal language, for example, standard procedure new uses to perform explicit allocation. Execution of new (p_1) allocates storage for the object type pointed to by p_1 and newly allocated object is left pointed by p_1 . In most implementations of Pascal, deallocation is completed by calling dispose. When evaluation of an expression results in storage being obtained to hold the value of the expression, implicit allocation occurs. In Lisp language, when cons is used, allocates a cell in a list. Cells automatically reclaim when that can no longer be arrived. Snobol allows the length of a string to vary at run time and manages the space required to hold the string in a heap.

Example – The Pascal program builds a linked list and prints the integers in the cells of link list. Its output is

77	4
5	3
6	2

Storage for the pointer head is in the activation record for the complete program, when execution begins at the line 15 of the program

new (p_1); $p_1 \uparrow \text{key} := k$; $p_1 \uparrow \text{info} := i$;
the call new (p_1) results in a cell being allocated somewhere within the heap; $p_1 \uparrow$ refers to this cell in the assignments. Cell allocated using new during an

activation of insert are held when control returns to the main program from the activation.

```

program table (input, output);
type link = ↑cell;
cell = record
    key, info : integer;
    next : link
end;

```

```

var head : link ;
procedure insert (k, i : integer);
var p1 : link;
begin
    new (p1); p1↑.key := k;
    p1↑.info := i;
    p1↑.next := head;
    head := p1
end;
begin
    head := nil
    insert (6, 2); insert (5, 3);
    insert (77, 4);
    writeln (head↑.key, head↑.info);
    writeln (head↑.next↑.key, head↑.next↑.info);
    writeln (head↑.next↑.next↑.key, head↑.
        next↑.next↑.info)
end.

```

Fig. 3.20 Dynamic Allocation of Cells using New in Pascal

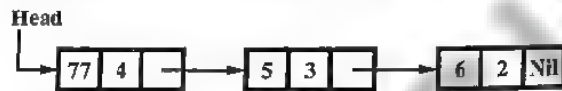


Fig. 3.21 Linked List Built by Fig. 3.20

Garbage – Dynamically allocated storage cannot become reachable. Storage that a program allocates but cannot refer to is known as garbage.

```

insert (6, 2); insert (5, 3); insert (77, 4); head↑.next := nil;
writeln (head↑.key, head↑.info);

```

The leftmost cell will now contain a nil pointer rather than a pointer to middle cell. The middle and rightmost cells become garbage, if the pointer to the middle cell is lost. Lisp language performs garbage collection. Pascal and C do not provide a feature of garbage collection.

Dangling References – Explicit deallocation can cause of an additional complication just like dangling references. When storage that has been deallocated is referred, a dangling reference occurs.

For example – Consider the effect of executing dispose (head↑.next).

```

insert (6, 2);
insert (5, 3);
insert (77, 4);
dispose (head↑.next);
writeln (head↑.key, head↑.info);

```

The cell to dispose deallocates the cell pointed to by head. However, head↑.next has been unchanged, so it is a dangling pointer to deallocated storage.

The concepts of dangling references and garbage can be expressed as if deallocation occurs before the last reference, it is called dangling reference and if deallocation occurs after the last reference, it is called garbage.

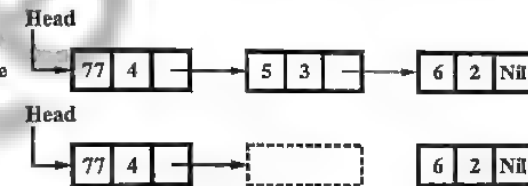


Fig. 3.22 Example of Dangling Reference and Garbage

Q.32. Discuss heap management and garbage collection with an example. (R.G.P.V., May 2019)

Ans. Refer to Q.22 (iii) and Q.31.

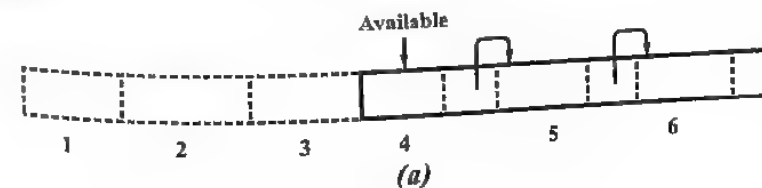
Q.33. Explain in detail different dynamic storage allocation strategies. (R.G.P.V., Dec. 2013)

Or

Explain briefly dynamic storage allocation. (R.G.P.V., May 2018)

Ans. The technique used to implement dynamic storage allocation depends on how storage is deallocated. If deallocation is implicit, then the run-time support package is responsible for determining when a storage block is no longer needed. The dynamic storage allocation techniques are categorized as follows –

(i) **Explicit Allocation of Fixed Sized Blocks** – The simplest form of dynamic allocation involves blocks of a fixed size. Allocation and deallocation can be done quickly by linking the blocks in a list with little or no storage overhead.



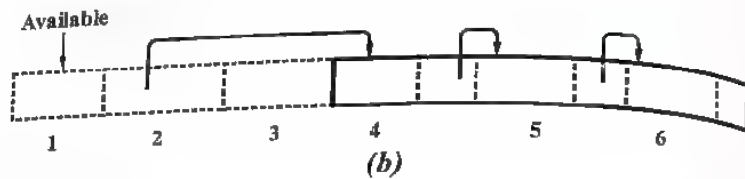


Fig. 3.23 A Deallocated Block is Added to the List of Available Blocks

Suppose that blocks are to be drawn from a contiguous area of storage. Initialization of the area is done by using a portion of each block for a link to the next block. Allocation consists of taking a block off the list and deallocation consists of putting the block back on the list. Each block is treated as a variable record. Block consists of link to the next block and the user program. There is no space overhead because the user program can use the entire block. When the block is returned, then the compiler routines use some of the space from the block itself to link it into the list of available blocks.

(ii) **Explicit Allocation of Variable Sized Blocks** – When blocks are allocated and deallocated, storage can become fragmented, i.e., the heap may consist of alternate blocks that are free and in use.

One method for allocating variable sized blocks is called the first-fit method. When a block of size s is allocated, the first free block is searched that is of size $f \geq s$. This block is then subdivided into a used block of size s and a free block of size $f - s$.



Fig. 3.24 Free and used Blocks in a Heap

When a block is deallocated, we check to see if it is next to a free block. The deallocated block is combined with a free block next to it to create a larger free block. There are also several tradeoffs between time, space and availability of large blocks.

(iii) **Implicit Deallocation** – Implicit deallocation requires co-operation between the user program and the run-time package. This co-operation is implemented by fixing the format of storage blocks. The first problem is that of recognizing block boundaries. The second problem is that of recognizing if a block is in use. Two approaches can be used for implicit deallocation –

(a) **Reference Counts** – Refer to Q.27 (iii).

(b) **Marking Techniques** – An alternative approach is to suspend temporarily execution of the user program and use the frozen pointers to determine which blocks are in use. This approach requires all the pointers into the heap to be known. All the blocks are marked unused. Then, pointers

are followed marking as used any block that is reached in the process. A final sequential scan of the heap allows all blocks still marked unused to be collected.

With variable sized blocks, the possibility of moving used storage blocks from their current positions. This process, called compaction moves all used blocks to one end of the heap, so that all the free storage can be collected into one large free blocks.



Fig. 3.25 Garbage Cells With Nonzero Reference Counts

Q.34. What are the difficulties faced by memory allocation for variable length requirements? Under what circumstances does external fragmentation happen? (R.G.P.V., Dec. 2017)

Ans. The difficulties faced by memory allocation for variable length requirements are as follows –

- Programmer has to deallocate the allocated memory manually.
- System will perform slow if allocated memory space is not released.
- Memory can be fragmented if memory allocation is not used properly.
- System can crash if the memory space is freed before the completion of task.
- Allocated memory space cannot be allocated to another program until it gets free.

Also refer to Q.27 (i) and Q.33 (ii).

Q.35. Write short note on symbol tables organization. (R.G.P.V., June 2003, Dec. 2003, June 2004, 2005, 2006, Dec. 2010, June 2011)

Or

What is symbol table? How is it used in compilers? (R.G.P.V., Dec. 2015)

Or

Write short note on symbol table. (R.G.P.V., Dec. 2017)

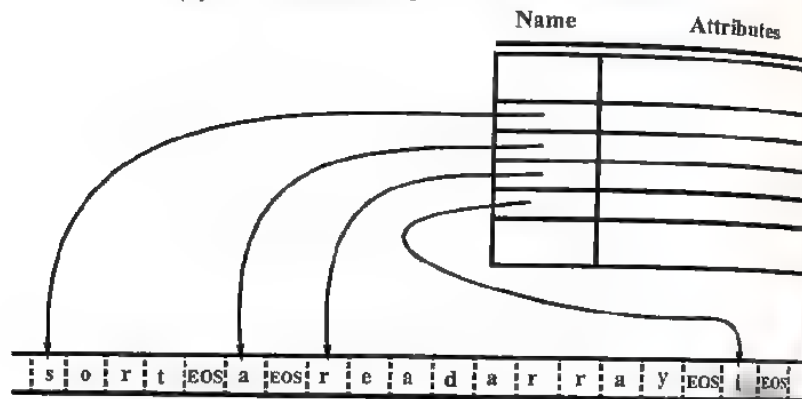
Or

Explain the symbol table management system. (R.G.P.V., Nov. 2018)

Ans. Compiler collects and uses the scope and binding information about the names that appears in the source program. This information is entered into a data structure called a symbol table. The name information includes the string of characters by which it is denoted, its type (e.g., integer, real, string), its form (e.g., a simple variable, a structure), its location in memory and other attributes depending on the language. Each entry in the symbol table is pair of the form (name, information). A symbol table is searched every time a name is encountered in the source text. Changes to the table occur if a new name or new information about an existing name is found. Information about that name is entered into the table during lexical and syntactic analysis.

Name	Attributes
s o r t	
a	
r e a d a r r a y	
i	

(a) In Fixed-size Space within a Record



(b) In a Separate Array

Fig. 3.26 Storing the Characters of a Name

The information collected in the symbol table is used during several stages in the compilation process. It is used in semantic analysis, i.e., in checking that uses of names are consistent with their implicit or explicit declarations. It is also used during code generation. The two symbol table mechanisms are linear lists and hash tables. Both mechanisms can be adapted readily to handle the most closely nested scope rule. The symbol table can be used to aid in error detection and correction.

It is useful for a compiler to be able to grow the symbol table dynamically, if necessary, at compile time. If the size of the symbol table is fixed when the compiler is written, then the size must be chosen large enough to handle any source program.

Symbol Table Contents – A symbol table is a table with two fields, a name field and an information field. The names in the symbol table denote objects of various sorts. There is separate tables for variable names, labels, procedure names, constants, field names and other types of names keywords are entered into the table initially, before lexical analysis begin. Attributes of a name are entered in response to declaration, which may be implicit. Labels are identifiers followed by colon so one action associated with recognizing such as an identifier may be to enter this fact into the symbol table. The syntax of procedure declaration specifies that certain identifiers are formal parameters.

Symbol Table Records and Names – The simplest way to implement a symbol table is as a linear array of records, one record per name. A record normally consists of a known number of consecutive words of memory. A common representation of a name is a pointer to the symbol table entry for it. In the record for a name, a pointer is placed to a separate arrays of the characters giving the position of the first character of the lexeme. The complete lexeme constituting a name must be stored to ensure that all uses of the same name can be associated with the same symbol table record.

Storage Allocation Information – Information about the storage locations that will be bound to names at runtime is kept in the symbol table. First considering the names with the static storage. If the target code is assembly language then scan the symbol table, after generating assembly code for the program, and generate assembly language data definitions to be appended to the executable portion of the assembly language program.

If the machine code is to be generated by the compiler, then the position of each data object relative to a fixed origin such as the beginning of an activation record must be ascertained. The same remark applies to blocks of data loaded as a module separate from the executable program.

Q.36. Explain in brief the various data structures that can be used in symbol table management. (R.G.P.V., Dec. 2005, June 2007, 2008)

Or

Explain the various data structures used in symbol table.

(R.G.P.V., Dec. 2007, 2009)

Or

Write a short note on data structures used in symbol table.

(R.G.P.V., Dec. 2008)

Or

Explain the various strategies of symbol table creation and organization.

(R.G.P.V., Dec. 2016)

Ans. During compilation the symbol table is searched every time an identifier is encountered. Data are added if a new name or new information about an existing name is found.

The various methods of symbol table management are as follows –

(i) **Linear List** – A linear list of records is the easiest way to implement a symbol table. The new names are added to the table in the order that they arrive. Whenever a new name is to be added to the table, the table is first searched linearly or sequentially to check whether or not the name is already present in the table. If the name is not present, then the record for new name is created and added to the list at a position specified by the available pointer, as shown in the fig. 3.27.

To retrieve the information about the name, the table is searched sequentially, starting from the first record in the table. The average number of comparisons, p required for search are

$$p = \frac{(n+1)}{2}$$

for successful search and

$$p = n$$

for an unsuccessful search, where n is the number of records in symbol table. The advantage of this organization is that it takes less space, and additions to the table are simple. The disadvantage of this method is that it has a higher accessing time.

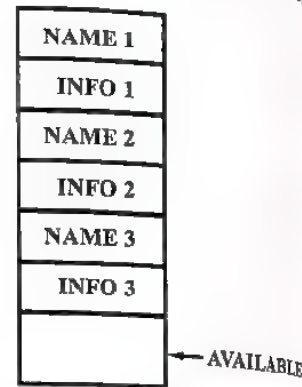


Fig. 3.27 A New Record is Added to the Linear List of Records

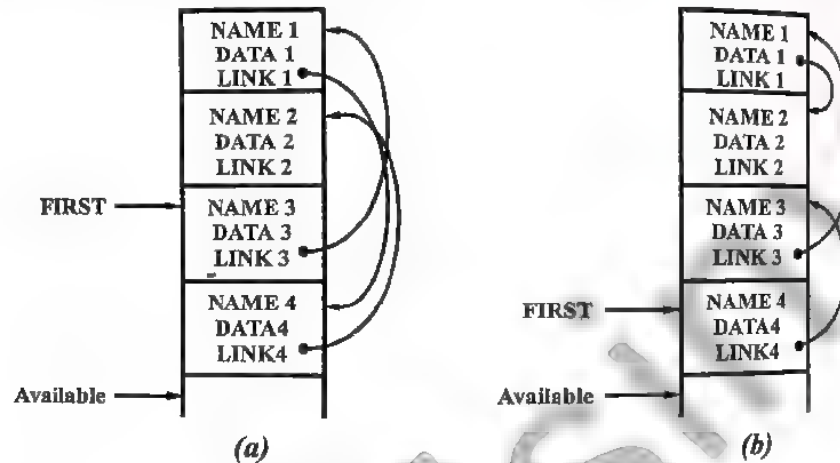


Fig. 3.28 Self Organizing Symbol Table

Self organizing list is a special case of the linear list in which a link field is added to each record in the simple list. In this searching is done in the order indicated by links. The first denotes the FIRST record on the linked list.

(ii) **Search Tree** – A search tree is a more efficient approach to symbol table organization. We add two links, left and right, in each record and these links point to the record in the search tree. Whenever a name is to be added, first the name is searched in the tree. If it does not exist, then a record for the new name is created and added at the proper position in the search tree. This organization has the property of alphabetical accessibility; that is, all the names accessible from name i will, by following a left link, precede name i in alphabetical order. Similarly, all the names accessible from name i will follow name i in alphabetical order by following the right link (see fig. 3.29). The

expected time needed to enter n names and to make m queries is proportional to $(m + n) \log_2 n$. So for greater number of records (higher n) this method has advantages over linear list organization.

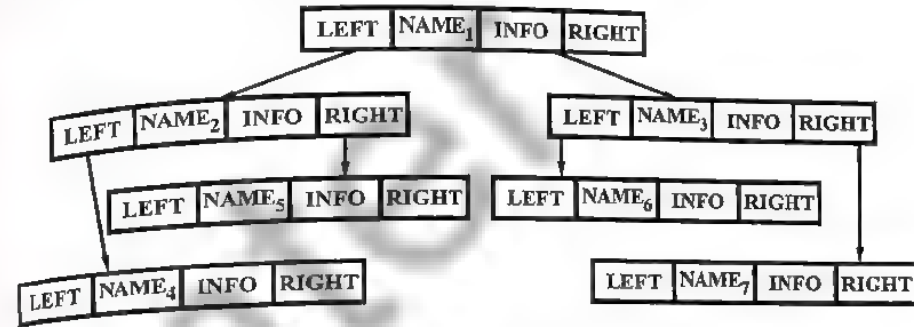


Fig. 3.29 The Search Tree Organization Approach to Symbol Table

(iii) **Hash Table** – A hash table is a table of k pointers numbered from zero to $k - 1$ that point to the symbol table and a record within the symbol table. To enter a name into symbol table, we find out the hash value of the name by applying a suitable hash function. The hash function maps the name into an integer between zero and $k - 1$, and using this value as an index in the hash table, we search the list of the symbol table records that is built on the hash index. If the name is not present in that list, we create a record for name and insert it at the head of the list. When retrieving the information associated with the name, the hash value of the name is first obtained, and the list that was built on this hash value is searched for information about the name (see fig. 3.30).

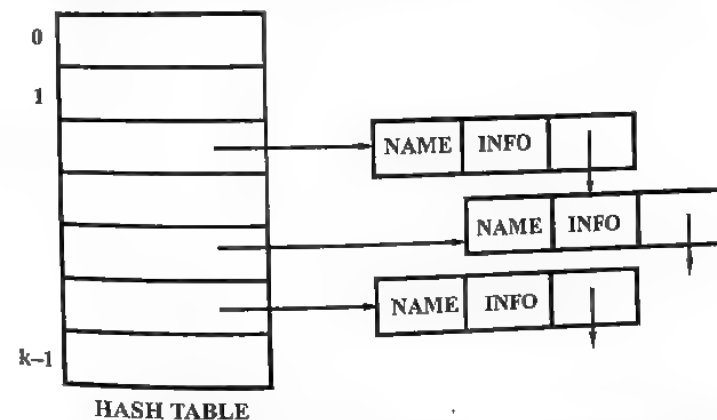


Fig. 3.30 Hash Table Method of Symbol Table Organization

Q.37. What is a symbol table? Explain how the symbol table in a compiler can be implemented by a hash table. (R.G.P.V., May 2018)

Ans. Refer to Q.35 and Q.36 (iii).

Q.38. Compare various symbol table management techniques.

(R.G.P.V., Dec. 2008)

Ans. The comparison of various symbol table management techniques as follows -

S. No.	Characteristics	Linear List	Search Tree	Hash Table
(i)	Algorithm	Simple	Complex	More complex
(ii)	Performance	Poor	Better	Best
(iii)	To enter n names and m inquiries, the total work is	$C_n(n + m)$ $C = \text{constant}$ representing the time necessary for a new machine operation.	$(m + n) \log_2 n$	$\frac{n(n + m)}{k}$, $k = \text{any constant}$, k can be made as large as we like
(iv)	Addition	The new names are added to the end of the list.	The new names are added at its proper position.	New names are added to the head of the list.
(v)	Retrieval	To retrieve the information, the table is searched sequentially starting from the first record.	The records are retrieved according to alphabetical order.	The records are retrieved through its hash value.
(vi)	Alphabetically accessing	Not present	Present	Not present
(vii)	Comparison for search	$(n + 1)/2$ for successful search, n for unsuccessfully search	$m \log_2 n$	Less than both linear list and search tree.
(viii)	Space required	It takes less space.	It takes more space in comparison to linear list.	It takes more space than both linear list and search tree.
(ix)	Accessing	Higher	Less	Less

Q.39. Discuss the symbol table organization, also give the difference between binary tree and hashing organization of symbol table.

(R.G.P.V., Dec. 2014)

Ans. Refer Q.35 and Q.38.

Q.40. Explain the various data structures used for implementing the symbol table and compare them.

(R.G.P.V., June 2010, Dec. 2011, June 2012)

Ans. Refer to Q.36 and Q.38.

Q.41. What is hashing? What are different types of hashing techniques available?

(R.G.P.V., June 2008, Dec. 2010)

Ans. Symbol table is a set of table entries, (K, V). Each entry contains a unique key, K, and a value (information), V. Each key uniquely identifies its entry. For searching table, a search key, k, is given and the table entry, (K, V) is to be found. Once the entry (K, V) is found its value V may be updated, it may be updated, retrieved or the entire entry, (K, V), may be removed from the table. If no entry with key K exists in the table, a new entry with K as its key may be inserted to the table.

Hashing is a technique of storing values in the tables and searching for them in linear, $O(n)$, worst case and extremely fast, $O(1)$, average-case time.

Hashing computes an integer, called the hash code, for each object. The computation is called the hash function, $h(k)$. It maps objects (e.g., keys K) to the array indices (e.g., 0, 1, ..., i max). An object with a key K has to be stored at location $h(K)$. The hash function must always return a valid index for the array.

The basic features of hashing are as follows -

Perfect Hash Function - A different index value for every key. But such a function can't be always found.

Collision - If two distinct keys, K_1, K_2 , map to the same table address, $h(K_1) = h(K_2)$.

Collision Resolution Policy - How to find additional storage to store one of the collided table entries.

Load Factor λ - Fraction of the already occupied entries.

There are four basic methods for choosing a hash function - division, folding, middle-squaring and truncation.

The main principal criterion for choosing a hash function is that it should be easy and quick to compute and it should achieve an even distribution of the keys over the possible array indices so that it should produce as few collisions as possible. It is possible to construct hash function which will generate a uniform distribution of indices only if we know in advance the set of keys which will occur.

ERROR DETECTION AND RECOVERY, AD-HOC AND SYSTEMATIC METHODS

Q.42. Explain error recovery in lexical analyzer. (R.G.P.V., Dec. 2004)

Or

What are error recovery action in lexical analysis phase?

(R.G.P.V., June 2003)

Ans. A lexical analyzer gets a very localized view of source program. If any string 'fi' is encountered then the analyzer cannot tell it is a misspelled of the keyword 'if' and will simply pass it as an identifier.

In a situation when lexical analyzer is unable to proceed as because none of the tokens matches a prefix of the remaining input then it opts for a panic mode recovery where successive characters are deleted from the remaining input until a well formed token is detected. In an interactive computing environment such a technique is quite adequate.

Other possible error-recovery actions are –

- (i) Deleting an extraneous character.
- (ii) Inserting a missing character.
- (iii) Replacing an incorrect character by a correct character.
- (iv) Transposing two adjacent characters.

Error transformation like these may be tried in an attempt to repair the input. The strategy is to see whether a prefix of the remaining input can be transformed into a valid lexeme by just a single error transformation. One way of finding the errors in a program is to compute the minimum number of error transformations required to transform the erroneous program into one which is syntactically correct.

However the lexical analyzer can only partially detect the errors in the following phases of the compiler also implement many error detecting and recovery routine.

Q.43. Explain the functions of a syntax analyzer.

Ans. There are various functions of a syntax analyzer. It checks the syntax of series of tokens received from the output of lexical analyzer for its validity. It outputs an error message where detects an error in the program and also do some error recovery so that further analysis can be carried. These functions of syntax analyzer are explained below –

(i) **Error Reporting/Handling** – As every language has a grammar so it has some syntaxes based on which proper programs are designed. The

programmer while writing a program do make mistakes, thus violating these syntaxes. The errors caused should be detected by the language compiler. The common errors which are encountered comes under any of the following category –

- (a) **Lexical Errors** – Such as mis-spelling and identifier, keyword or operator.
- (b) **Syntactic Errors** – Such as an arithmetic expression with unbalanced parentheses.
- (c) **Semantic Errors** – Such as an operator applied to an incompatible operand.
- (d) **Logical Errors** – Such as an infinitely recursive call.

The error detection and recovery in a compiler held in syntax analysis phase because many errors are syntactic in nature or exposed when the stream of tokens coming from lexical analyser violates the grammatical rules. As the parsing is done by syntax analyser. The parser itself handles the syntactic errors such that the following objectives are attained –

- (a) All errors should be reported clearly and accurately.
- (b) An error recovery routine should be incorporated so as to be able for detecting subsequent errors.
- (c) The processing of correct programs should not be affected or even slowed down during error handling.

Several parsing methods, like LL and LR methods, detect an error as soon as possible. Error detection is complicated process sometimes it occurs much earlier than the place where it is first recognized. The parsing methods have *viable-prefix property*.

(ii) **Error Recovery** – There are the following strategies that a parser can employ to recover from a syntactic error.

(a) **Panic Mode** – This is the simplest method to implement and can be used by most parsing methods, when an error is found the parser discards input symbols until one of a designated set of synchronizing tokens is found. The synchronizing tokens are usually delimiters, such as semicolon or end. It skips a considerable amount of input without checking it for additional errors, it does not go into an infinite loop.

(b) **Phrase Level** – In this method, some local correction is made on the remaining input, to correct the error, such as replacing a comma by a semicolon, deleting an extraneous semicolon, or inserting a missing semicolon.

(c) **Error Production** – Some errors hinders the construction of parse tree, if a programmer is aware of common errors. The grammar is

augmented for language with some error productions. Whenever the parser make use of these production it also generates proper reporting however parsing continues.

(d) **Global Correction** – These corrections help a parser to change the wrong input string encountered to a valid string with minimal change as there are predesigned algorithm to achieve the same. This recovery process is very costlier and often changes the actual program.

Q.44. What do you mean by Adhoc and systematic methods.

Ans. Refer to Q.43 (ii).

UNIT

4

CODE GENERATION

INTERMEDIATE CODE GENERATION – DECLARATIONS, ASSIGNMENT STATEMENTS, BOOLEAN EXPRESSIONS, CASE STATEMENTS, BACK PATCHING, PROCEDURE CALLS

Q.1. What is intermediate code ?

Ans. The intermediate code is a translated form of the input source stored in some data structure like array, tree, etc. The back end generates the target code by traversing this data structure. The intermediate code is also called as intermediate representation (IR). There are two commonly used intermediate code forms in compilers –

- (i) Three address code (TAC)
- (ii) Abstract syntax tree (AST).

Q.2. Explain briefly three address code. (R.G.P.V., Dec. 2015)

Or

Write short note on three address code. (R.G.P.V., Nov. 2018)

Ans. Three address code is a sequence of statements of the general form –

$$A := B \text{ op } C$$

where, A, B and C are programmer-defined names, constants or compiler generated temporary names; op stands for any operator such as fixed or floating point arithmetic operator, or a logical operator on boolean valued data. The three address code as the name suggests that each of the statement usually contains three addresses, two for the operands and one for the result. It does not permit built-up or complicated arithmetic expression, as there is only one operator on the right side of a statement. Thus an expression like $x + y * z$ would be translated into a sequence

$$\begin{aligned} t_1 &:= y * z \\ t_2 &:= x + t_1 \end{aligned}$$

where t_1 and t_2 are compiler generated temporary names. The use of names for the intermediate values computed by a program allows TAC to be easily rearranged unlike postfix notation.

Q.3. What is a three address code ? What are its types ? How is it implemented ?
(R.G.P.V., June 2016)

Ans. Refer to Q.2.

Statements can have symbolic labels and there are statements for flow of control. A symbolic label represents the index of a three address code in the array holding intermediate code. There are the common three address statements –

(i) Assignment statements of the form $A := B \text{ op } C$, where op is a binary arithmetic or logical operator.

(ii) Assignment instructions of the form $a := \text{op } B$, where op is a unary operation. Essential unary operations include unary minus, logical negation, shift operators, and conversion operators that, for example, convert a fixed point number to a floating point number.

(iii) Copy statements of the form $A := B$ where the value of B is assigned to A.

(iv) The unconditional jump goto L. The three address statement with label L is the next to be executed.

(v) Conditional jumps such as if $A \text{ rel op } B$ goto L. This instruction applies a relational operator rel op ($<$, $=$, \geq , etc.) to A and B and executes the statement with L next if A stands in relation rel op to B. If not, the three address statement following if $A \text{ rel op } B$ goto L is executed next as in the usual sequence.

(vi) Param A and call P, n. These instructions are used to implement a procedure call. Their typical use is as the sequence of three address statements –

Param A_1

Param A_2

.....

Param A_n

call P, n

Generated as part of a call of procedure P (A_1, A_2, \dots, A_n). The integer n indicating the number of actual parameters in "call P, n" is not redundant because calls can be nested.

(vii) Indexed assignments of the form $A := B[I]$ and $A[I] := B$. The first of these sets A to the value in the location I memory units beyond location B. $A[I] := B$ sets the contents of the location I units beyond A to the value of B. In both these instructions A, B and I refer to data objects will be represented by pointers to the symbol table.

(viii) Address and pointer assignments of form $A := \text{addr } B$, $A = *B$, and $*A = B$. The first of these sets the value of A to the location of B. B is a name which denotes an expression with an I-value such as $X[I, J]$. A is a

pointer name or temporary. That is, the r-value of A is the l-value of something else. In $A := *B$, B is a pointer or a temporary whose r-value is a location. The r-value of A is made equal to the content of that location. Finally, $*A := B$ sets the r-value of the object pointed to by A to the r-value of B.

Implementation of Three Address Statements –

A three address statement can be implemented as records with fields for the operator and the operands. The representations are quadruples, triples and indirect triples.

Quadruples – It is a record structure with four fields like op, arg1, arg2 and result. The op contains an internal code for the operator. For example, $x := y \text{ op } z$ is represented by putting y in arg1, z in arg2 and x in result. Statements with unary operators do not use arg 2. Conditional and unconditional jumps put the target label in result.

The content of fields arg1, arg2 and result are pointers to the symbol table entries for the names represented by these fields.

For example, an assignment statement $A := -B * (C + D)$ can be translated to three address statements

$t_1 := -B$

$t_2 := C + D$

$t_3 := t_1 * t_2$

$A := t_3$

	op	arg1	arg2	result
(0)	u minus	B		t_1
(1)	+	C	D	t_2
(2)	*	t_1	t_2	t_3
(3)	:=	t_3		A

Fig. 4.1 Quadruple Representation of TAC

	op	arg1	arg2
(0)	u minus	B	
(1)	+	C	D
(2)	*	(1)	(1)
(3)	:=	A	(2)

Fig. 4.2 Triple Representation of TAC

Triples – To avoid entering temporary names into the symbol table, a temporary value by the position of statement that computes it is referred. In this the three address statement can be representable by a structure with only three fields op arg1, arg2. The two arguments are either pointers to the symbol table or pointers into the structure itself. The intermediate code format with three fields is known as triples.

Indirect Triples – Three address code which has been considered is that of listing pointers to triples, rather than listing the triples themselves. This type of implementation is known as indirect triples.

	STATEMENT		op	arg1	arg2
(0)	(14)	(14)	u minus	B	
(1)	(15)	(15)	+	C	D
(2)	(16)	(16)	*	(14)	(15)
(3)	(17)	(17)	:=	A	(16)

Fig. 4.3 Indirect Triples Representation of TAC

Q.4. Explain quadruple, triple and indirect triple with suitable example.
(R.G.P.V., Dec. 2010)

Ans. Refer to Q.3.

Q.5. How can three address code be implemented in a compiler? Describe triples and indirect triples method of implementing TAC with example.
(R.G.P.V., May 2018)

Ans. Refer to Q.2 and Q.3.

Q.6. What is an intermediate code? What are the benefits of intermediate code generation? How can three address code be implemented in a compiler? Describe quadruple.
(R.G.P.V., May 2018)

Ans. Intermediate Code – Refer to Q.1.

Benefits – The benefits of intermediate code generation are as follows-

- A compiler for different machines can be created by attaching different backend to the existing front ends of each machine.
- A compiler for different source languages (on the same machine) can be created by providing different front ends for corresponding source language to existing backend.
- A machine independent code optimizer can be applied to intermediate code in order to optimize the code generation.

Implementation of Three Address Code – Refer to Q.2 and Q.3

Q.7. Discuss the declaration of variables in intermediate code generation

Ans. The syntax of languages such as C, Pascal and Fortran, allows all the declarations in a single procedure to be processed as a group. In the case a global variable, say *offset*, can keep track of the next available relative address.

In the translation scheme of fig. 4.4 nonterminal P generates a sequence of declarations of the form *id : T*. Before the first declaration is considered *offset* is set to 0. As each new name is seen, that name is entered in the symbol table with *offset* equal to the current value of *offset* and *offset* is incremented by the width of the data object denoted by that name.

The procedure *enter(name, type, offset)* creates a symbol table entry for *name*, gives it type *type* and relative address *offset* in its data area. We use synthesized attributes *type* and *width* for nonterminal *T* to indicate the type and width, or number of memory units taken by objects of that type. If type expressions are represented by graphs, then attribute *type* might be a pointer to the node representing a type expression.

$P \rightarrow D$	{ <i>offset</i> := 0 }
$D \rightarrow D ; D$	{ <i>enter</i> (<i>id.name</i> , <i>T.type</i> , <i>offset</i>); <i>offset</i> := <i>offset</i> + <i>T.width</i> }
$D \rightarrow id : T$	
$T \rightarrow integer$	{ <i>T.type</i> := integer; <i>T.width</i> := 4 }
$T \rightarrow real$	{ <i>T.type</i> := real; <i>T.width</i> := 8 }
$T \rightarrow array [num] of T_1$	{ <i>T.type</i> := array(<i>num.val</i> , <i>T_1.type</i>); <i>T.width</i> := <i>num.val</i> × <i>T_1.width</i> }
$T \rightarrow \uparrow T_1$	{ <i>T.type</i> := pointer(<i>T_1.type</i>); <i>T.width</i> := 4 }

Fig. 4.4 Computing the Types and Relative Addresses of Declared Names

In fig. 4.4, integers have width 4 and reals have width 8. The width of an array is obtained by multiplying the width of each element by the number of elements in the array. The width of each pointer is assumed to be 4. In Pascal and C, a pointer may be seen before we learn the type of the object pointed to. Storage allocation for such types is simpler if all pointers have the same width.

Q.8. Compare the relative merits and demerits of the following –
(i) Quadruples (ii) Triples (iii) Indirect triples.

Ans. The triples and quadruples differ on the basis of how much indirection is present in their representation. When we produce object code, each datum, temporary or programmer-defined, will be assigned some memory location. This location will be placed in the symbol-table entry for the datum. Using the quadruple notation, the location for each temporary can be immediately accessed through symbol table, from where it is needed. If the triples notation is used, it has no idea, unless scan the code, how many words must be allocated for temporaries; so with triples the assignment of locations to temporaries is usually referred to code generation.

An important benefit of quadruples is an optimizing compiler. Using the quadruple notation, the symbol table interposes an extra degree of indirection between the computation of a value and its use. However, in the triples notation, moving a statement that defines a temporary value requires us to change all pointers to that statement in the *arg1* and *arg2* arrays. This problem makes triples difficult to use in an optimizing compiler.

Indirect triples present no such problem. To move a statement we simply reorder the STATEMENT list. Since pointers to temporary values refer to the op-arg1 -arg2 arrays, which are not changed. Thus, indirect triples look very much like quadruples.

The problem of quadruples is that they tend to clutter up the symbol table with temporary names. If we use integer codes for temporaries and do not enter temporaries in the symbol table, triples have the same problem when we assign locations to temporaries.

Q.9. Compare the relative merits and demerits of the following –

(i) Quadruples (ii) Triples (iii) Indirect triples
and write the quadruples, triple and indirect triples for the expression –
 $-(a * b) + (c + d) - (a + b + c + d)$

(R.G.P.V., Dec. 2002, June 2011)

Ans. Merits and Demerits – Refer to Q.8.

The three address code of the given expression is as follows –

$$\begin{aligned} t_1 &= a * b & t_2 &= -t_1 \\ t_3 &= c + d & t_4 &= t_2 + t_3 \\ t_5 &= a + b & t_6 &= t_5 + c \\ t_7 &= t_6 + d & t_8 &= t_4 - t_7 \end{aligned}$$

The quadruple of the given expression is shown below –

	Operator	Operand1	Operand2	Result
(1)	*	a	b	t ₁
(2)	–	t ₁		t ₂
(3)	+	c	d	t ₃
(4)	+	t ₂	t ₃	t ₄
(5)	+	a	b	t ₅
(6)	+	t ₅	c	t ₆
(7)	+	t ₆	d	t ₇
(8)	–	t ₄	t ₇	t ₈

The triple of the given expression is shown below –

	Operator	Operand1	Operand2
(1)	*	a	b
(2)	–	(1)	
(3)	+	c	d
(4)	+	(2)	(3)
(5)	+	a	b
(6)	+	(5)	c
(7)	+	(6)	d
(8)	–	(4)	(7)

The indirect triple of the given expression is –

	Statement		Operator	Operand1	Operand2
(1)	(10)	(10)	*	a	b
(2)	(20)	(20)	–	(10)	
(3)	(30)	(30)	+	c	d
(4)	(40)	(40)	+	(20)	(30)
(5)	(50)	(50)	+	a	b
(6)	(60)	(60)	+	(50)	c
(7)	(70)	(70)	+	(60)	d
(8)	(80)	(80)	–	(40)	(70)

Q.10. How can we keep track of scope information ?

Ans. In a language with nested procedures, names local to each procedure can be assigned relative addresses using the approach of fig. 4.4. When a nested procedure is seen, processing of declarations in the enclosing procedure is temporarily suspended. This approach will be illustrated by adding semantic rules to the following language

$P \rightarrow D$

$D \rightarrow D; D[id : T] \text{proc id}; D; S$

...(i)

The productions for nonterminals S for statements and T for types are not shown because we focus on declarations. The nonterminal T has synthesized attributes *type* and *width* as in the translation scheme of fig. 4.4.

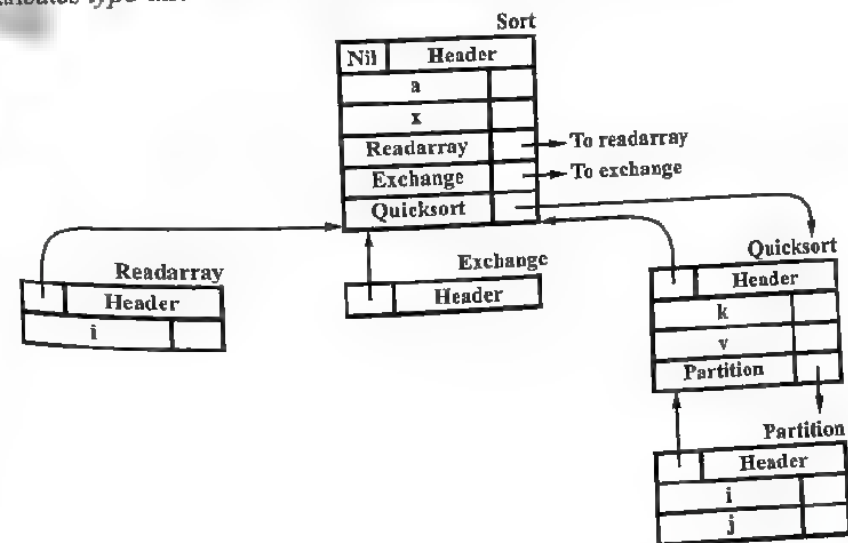


Fig. 4.5 Symbol Tables for Nested Procedures

A new symbol table is created when a procedure declaration $D \rightarrow \text{proc id } D_1; S$ is seen and entries for the declarations in D_1 are created in the new

table. The new table points back to the symbol table of the enclosing procedure. The name represented by *id* itself is local to the enclosing procedure. The only change from the treatment of variable declarations in fig. 4.4 is that the procedure *enter* is told which symbol table to make an entry in.

For example, symbol tables for five procedures are shown in fig. 4.5. The nesting structure of the procedures can be deduced from the links between the symbol tables. The symbol tables for procedures *readarray*, *exchange* and *quicksort* point back to that for the containing procedure *sort*, consisting of the entire program. Since *partition* is declared within *quicksort*, its table points to that of *quicksort*.

Q.11. Show that how names can be looked up in symbol table.

Or

Give the translation scheme for converting the assignments into three address code.
(R.G.P.V., Dec. 2013)

Ans. The semantic actions in fig. 4.7 use procedure *emit* to emit three address statements to an output file, rather than building up *code* attributes for nonterminals. From syntax-directed translation, translation can be done by emitting to an output file if the *code* attributes of the nonterminals on the left sides of productions are formed by concatenating the *code* attributes of the nonterminals on the right, in the same order that the nonterminals appear on the right side, perhaps with some additional strings in between.

By reinterpreting the *lookup* operation in fig. 4.7, the translation scheme can be used even if the most closely nested scope rule applies to nonlocal names, as in Pascal. For concreteness, suppose that the context in which an assignment appears is given by the following grammar –

$$\begin{aligned} P &\rightarrow M D \\ M &\rightarrow \epsilon \\ D &\rightarrow D ; D \mid id ; T \mid \text{proc } id ; N D ; S \\ N &\rightarrow \epsilon \end{aligned}$$

Nonterminal *P* becomes the new start symbol when these productions are added to those in fig. 4.7.

For each procedure generated by this grammar, the translation scheme in fig. 4.6 sets up a separate symbol table. Each such symbol table has a header containing a pointer to the table for the enclosing procedure. When the statement forming a procedure body is examined, a pointer to the symbol table for the procedure appears on top of the stack *tblptr*. This pointer is pushed onto the stack by actions associated with the marker nonterminal *N* on the right side of $D \rightarrow \text{proc } id ; N D ; S$.

$$\begin{aligned} P &\rightarrow M D && \{ \text{addwidth}(\text{top}(\text{tblptr}), \text{top}(\text{offset})); \\ M &\rightarrow \epsilon && \text{pop}(\text{tblptr}); \text{pop}(\text{offset}) \} \\ D &\rightarrow D_1 ; D_2 && \{ t := \text{mktable}(\text{nil}); \\ D &\rightarrow \text{proc } id ; N D_1 ; S && \text{push}(t, \text{tblptr}); \text{push}(0, \text{offset}) \} \\ &&& \{ t := \text{top}(\text{tblptr}); \\ &&& \text{addwidth}(t, \text{top}(\text{offset})); \\ &&& \text{pop}(\text{tblptr}); \text{pop}(\text{offset}); \\ &&& \text{enterproc}(\text{top}(\text{tblptr}), id.name, t) \} \\ D &\rightarrow id : T && \{ \text{enter}(\text{top}(\text{tblptr}), id.name, T.type, \text{top}(\text{offset})); \\ N &\rightarrow \epsilon && \text{top}(\text{offset}) := \text{top}(\text{offset}) + T.width \} \\ &&& \{ t := \text{mktable}(\text{top}(\text{tblptr})); \\ &&& \text{push}(t, \text{tblptr}); \text{push}(0, \text{offset}) \} \end{aligned}$$

Fig. 4.6 Processing Declarations in Nested Procedures

Let the productions for nonterminal *S* be those in fig. 4.7. Names in an assignment generated by *S* must have been declared in either the procedure that *S* appears in, or in some enclosing procedure. When applied to *name*, the modified *lookup* operation first checks if *name* appears in the current symbol table, accessible through *top(tblptr)*. If not, *lookup* uses the pointer in the header of a table to find the symbol table for the enclosing procedure and looks for the name there. If the name cannot be found in any of these scopes, then *lookup* returns *nil*.

$$\begin{aligned} S &\rightarrow id := E && \{ P := \text{lookup}(id.name); \\ &&& \text{if } p \neq \text{nil then} \\ &&& \text{emit}(p := E.place) \\ &&& \text{else error} \} \\ E &\rightarrow E_1 + E_2 && \{ E.place := \text{newtemp}; \\ &&& \text{emit}(E.place := E_1.place + E_2.place) \} \\ E &\rightarrow E_1 * E_2 && \{ E.place := \text{newtemp}; \\ &&& \text{emit}(E.place := E_1.place * E_2.place) \} \\ E &\rightarrow - E_1 && \{ E.place := \text{newtemp}; \\ &&& \text{emit}(E.place := \text{'minus'} E_1.place) \} \\ E &\rightarrow (E_1) && \{ E.place := E_1.place \} \\ E &\rightarrow id && \{ p := \text{lookup}(id.name); \\ &&& \text{if } p \neq \text{nil then} \\ &&& E.place := p \\ &&& \text{else error} \} \end{aligned}$$

Fig. 4.7 Translation Scheme to Produce Three-address Code for Assignments

Q.12. How can addressing array elements be accessed ?

Ans. Elements of an array can be accessed quickly if the elements are stored in a block of consecutive locations. If the width of each array element is *w*, then the *i*th element of array *A* begins in location

$$\text{base} + (i - \text{low}) \times w \quad \dots(i)$$

where *low* is the lower bound on the subscript and *base* is the relative address of *A[low]*.

The expression (i) can be partially evaluated at compile time if it is rewritten as

$$i \times w + (\text{base} - \text{low} \times w)$$

The subexpression $c = \text{base} - \text{low} \times w$ can be evaluated when the declaration of the array is seen. We assume that c is saved in the symbol table entry for A , so the relative address of $A[i]$ is obtained by simply adding $i \times w$ to c .

Compile-time precalculation can also be applied to address calculations of elements of multi-dimensional arrays. A two-dimensional array is normally stored in one of two forms either *row-major* (row-by-row) or *column-major* (column-by-column). Fig. 4.8 shows the layout of a 2×3 array A in (a) row-major form and (b) column-major form. Fortran uses column-major form; Pascal uses row-major form, because $A[i, j]$ is equivalent to $A[j, i]$, and the elements of each array $A[i]$ are stored consecutively.

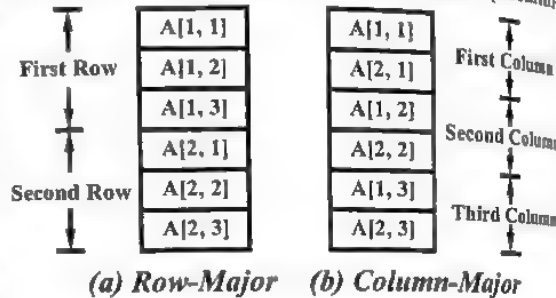


Fig. 4.8 Layouts for Two Dimensional Array

In the case of a two-dimensional array stored in row-major form, the relative address of $A[i_1, i_2]$ can be calculated by the formula –

$$\text{base} + ((i_1 - \text{low}_1) \times n_2 + i_2 - \text{low}_2) \times w$$

where low_1 and low_2 are the lower bounds on the values of i_1 and i_2 and n_2 is the number of values that i_2 can take. That is, if high_2 is the upper bound on the value of i_2 , then $n_2 = \text{high}_2 - \text{low}_2 + 1$. Assuming that i_1 and i_2 are the only values that are not known at compile time, we can rewrite the above expression as

$$((i_1 \times n_2) + i_2) \times w + (\text{base} - ((\text{low}_1 \times n_2) + \text{low}_2) \times w) \quad \dots(ii)$$

The last term in this expression can be determined at compile time.

Q.13. Suggest a suitable syntax directed translation scheme for array references. Using this scheme translate the following statement to intermediate code –

$$A(I, J) := B(I, J) + C[A(K, L)] + D(I, J)$$

Assume the necessary dimensions.

Or

Write a translation scheme to generate intermediate code for assignment statements with array references.

(R.G.P.V., Dec. 2005)

(R.G.P.V., June 2010)

Ans. An array reference is an expression with an l -value. Therefore, to capture its syntactic structure, we add the following productions to the

grammar;

$$L \rightarrow \text{id} [\text{elist}]$$

$$\text{elist} \rightarrow \text{elist}, E/E$$

An array reference in a source program is replaced by the l -value of an expression that specifies reference to an element of the array. Computing the l -value involves finding the offset of the referred element of the array and then adding it to the base. But since deriving an offset depends on the subscripts used in an array reference, and the values of these subscripts are not known during the compilation, unless the subscripts are constant expressions, a compiler has to generate the code for evaluating the l -value of an expression that specifies the reference to an element of an array. This l -value computation is achieved as follows –

$$l\text{-value}(a[i_1, i_2, i_3, \dots, i_k]) = \text{addr}(a) + \text{offset}$$

$$\text{offset} = [(i_1 - l_{b1})(ub_2 - l_{b2} + 1)(ub_3 - l_{b3} + 1)$$

$$\dots (ub_k - l_{bk} + 1) + (i_2 - l_{b2})(ub_3 - l_{b3} + 1)$$

$$(ub_4 - l_{b4} + 1)$$

$$\dots (ub_k - l_{bk} + 1) + \dots + (i_k - l_{bk})]^*$$

Where l_{bi} and ub_i are the lower and upper bounds of the i^{th} dimension. If the lower bound of each dimension is one, and the upper bound of the i^{th} dimension is d_i , then the offset computing formula becomes –

$$\text{offset} = [(i_1 - 1) \times d_2 \times d_3 \times$$

$$\dots \times d_k + (i_2 - 1) \times d_3 \times d_4 \times$$

$$\dots \times d_k + \dots + (i_k - 1) \times \text{bpw}]$$

$$\text{offset} = [i_1 \times d_2 \times d_3 \times \dots \times d_k + i_2 \times d_3 \times d_4 \times \dots \times d_k + \dots + i_k] \times \text{bpw}$$

The $[i_1 \times d_2 \times d_3 \times \dots \times d_k + i_2 \times d_3 \times d_4 \times \dots \times d_k + \dots + i_k] \times \text{bpw}$ is a variable part of the offset computation, whereas $[d_2 \times d_3 \times \dots \times d_k + d_3 \times d_4 \times \dots \times d_k + \dots + d_k] \times \text{bpw}$ is a constant part of the offset computation and is not required to be computed for every reference to an array a . It can be computed once while processing the declaration of the array a . We call this value 'constant C '. Therefore –

$$\text{offset} = V - C$$

where V is variable part, and

$$l\text{-value}(a[i_1, i_2, i_3, \dots, i_k]) = \text{addr}(a) + V - C$$

Since $\text{addr}(a)$ is fixed, we can combine C with $\text{addr}(a)$ and store this value in an attribute, $L.\text{place}$, and we can store V in another attribute, $L.\text{off}$ so that –

$$l\text{-value}(a[i_1, i_2, i_3, \dots, i_k]) = L.\text{place}[L.\text{off}].$$

Hence, the translation of an array reference involves generating code for computing V , and V is made a value of attribute $L.\text{off}$. We compute $\text{addr}(a) - C$ and make it the value of the attribute $L.\text{place}$.

Translation of Statement – The statement is,

$$A(I, J) := B(I, J) + C[A(K, L)] + D(I, J)$$

Suppose A, B and D are arrays of size 30×40 , and C is array of size 20. There are four bytes per word, and the arrays are allocated statically. When the above translation scheme is used to translate this construct, the three address code generated is –

$t_1 = I * 40$	$t_8 = \text{addr}(A) - 164$
$t_1 = t_1 + J$	$t_9 = t_8[t_7]$
$t_1 = t_1 * 4$	$t_9 = t_9 * 4$
$t_2 = \text{addr}(A) - 164$	$t_{10} = \text{addr}(C) - 4$
$t_3 = t_2[t_1]$	$t_{11} = t_{10}[t_9]$
$t_4 = I * 40$	$t_{12} = I * 40$
$t_4 = t_5 + J$	$t_{12} = t_{12} + J$
$t_4 = t_5 * 4$	$t_{12} = t_{12} * 4$
$t_5 = \text{addr}(B) - 164$	$t_{13} = \text{addr}(D) - 164$
$t_6 = t_6[t_5]$	$t_{14} = t_{13}[t_{12}]$
$t_7 = K * 40$	$t_{15} = t_6 + t_{11}$
$t_7 = t_7 + L$	$t_{15} = t_{15} + t_{14}$
$t_7 = t_7 * 4$	$t_3 = t_{16}$

Q.14. What are the methods of translating Boolean expressions?

Ans. There are two principal methods of representing the value of a Boolean expression. The first method is to encode true and false numerically and to evaluate a Boolean expression analogously to an arithmetic expression. Often 1 is used to denote true and 0 to denote false although many other encodings are also possible. For example, we could let any nonzero quantity denote true and zero denote false, or we could let any nonnegative quantity denote true and any negative number denote false.

The second principal method of implementing Boolean expression is by flow of control, that is, representing the value of a Boolean expression by a position reached in a program. This method is particularly convenient in implementing the Boolean expressions in flow-of-control statements, such as the *if-then* and *while-do* statements. For example, given the expression E_1 or E_2 , if we determine that E_1 is true, then we can conclude that the entire expression is true without having to evaluate E_2 .

The semantics of the programming language determines whether all parts of a Boolean expression must be evaluated. If the language definition permits portions of a Boolean expression to go unevaluated, then the compiler can optimize the evaluation of Boolean expressions by computing only enough of an expression to determine its value. Thus, in an expression such as E_1 or E_2 , neither E_1 nor E_2 is necessarily evaluated fully. If either E_1 or E_2 is an expression with side effects, then an unexpected answer may be obtained.

Q.15. Discuss about numerical representation of Boolean expressions.

Ans. We assume the implementation of Boolean expressions using 1 to denote true and 0 to denote false. Expressions will be evaluated completely, from left to right, in a manner similar to arithmetic expressions. For example, the translation for

a or b and not c

is the three-address sequence

$t_1 := \text{not } c$	
$t_2 := b \text{ and } t_1$	
$t_3 := a \text{ or } t_2$	
$E \rightarrow E_1 \text{ or } E_2$	{ E.place := newtemp; emit(E.place := E ₁ .place 'or' E ₂ .place)}
$E \rightarrow E_1 \text{ and } E_2$	{ E.place := newtemp; emit(E.place := E ₁ .place 'and' E ₂ .place)}
$E \rightarrow \text{not } E_1$	{ E.place := newtemp; emit(E.place := 'not' E ₁ .place)}
$E \rightarrow (E_1)$	{ E.place := E ₁ .place}
$E \rightarrow \text{id}_1 \text{ relop } \text{id}_2$	{ E.place := newtemp; emit('if' id ₁ .place relop.op id ₂ .place 'goto' nextstat + 3); emit(E.place := '0'); emit('goto' nextstat + 2); emit(E.place := '1')}
$E \rightarrow \text{true}$	{ E.place := newtemp; emit(E.place := '1')}
$E \rightarrow \text{false}$	{ E.place := newtemp; emit(E.place := '0')}

Fig. 4.9 Translation Scheme using a Numerical Representation for Booleans

A relational expression such as $a < b$ is equivalent to the conditional statement *if a < b then 1 else 0*, which can be translated into the three-address code sequence –

```

100 : if a < b goto 103
101 : t := 0
102 : goto 104
103 : t := 1
104 :
```

A translation scheme for producing three-address code for Boolean expressions is shown in fig. 4.9. In this scheme, we assume that *emit* places three-address statements into an output file in the right format, that *nextstat* gives the index of the next three-address statement in the output sequence, and that *emit* increments *nextstat* after producing each three-address statement.

Q.16. What do you understand by short circuit code or jumping code of Boolean expression?

Ans. We can translate a Boolean expression into three address code without generating code for any of the Boolean operators and without having the code to evaluate the entire expression. This style of evaluation is sometimes called "short circuit" or "jumping" code. It is possible to evaluate Boolean expressions without generating code for the Boolean operators **and**, **or** and **not** if we represent the value of an expression by a position in the code sequence. For example, in fig. 4.10, we can tell what value t_1 will have by whether we reach statement 101 or statement 103, so the value of t_1 is redundant. For many boolean expressions, it is possible to determine the value of the expression without having to evaluate it completely.

100: if $a < b$ goto 103	107: $t_2 := 1$
101: $t_1 := 0$	108: if $e < f$ goto 111
102: goto 104	109: $t_3 := 0$
103: $t_1 := 1$	110: goto 112
104: if $c < d$ goto 107	111: $t_3 := 1$
105: $t_2 := 0$	112: $t_4 := t_2$ and t_3
106: goto 108	113: $t_5 := t_1$ or t_4

Fig. 4.10 Translation of $a < b$ or $c < d$ and $e < f$

Q.17. Discuss the flow-of-control statements for Boolean expression.

Ans. We consider the translation of Boolean expressions into three address code in the context of *if-then*, *if-then-else*, and *while-do* statements such as those generated by the following grammar –

$S \rightarrow$ if E then S_1
 | if E then S_1 else S_2
 | while E do S_1

In each of these productions, E is the Boolean expression to be translated. In the translation, we assume that a three address statement can be symbolically labeled, and that the function *newlabel* returns a new symbolic label each time it is called.

With a Boolean expression E , we associate two labels – $E.true$, the label to which control flows if E is true, and $E.false$ the label to which control flows if E is false. The semantic rules for translating a flow-of-control statement S allow control to flow from the translation $S.code$ to the three address instruction immediately following $S.code$. In some cases, the instruction immediately following $S.code$ is a jump to some label L . A jump to L from within $S.code$ is avoided using an inherited attribute $S.next$. The value of $S.next$ is a label that is attached to the first three address instruction to be executed after the code for S .

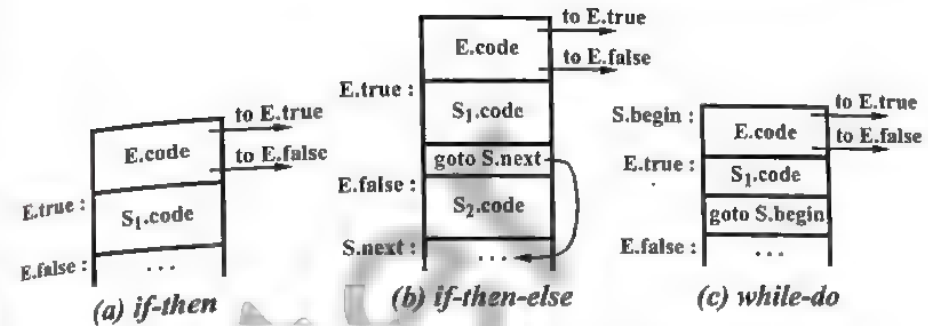


Fig. 4.11 Code for *if-then*, *if-then-else*, and *while-do* statements

In translating the *if-then* statement $S \rightarrow$ if E then S_1 , a new label $E.true$ is created and attached to the first three address instruction generated for the statement S_1 , as in fig. 4.11 (a). The code for E generates a jump to $E.true$ if E is true and a jump to $S.next$ if E is false. We therefore set $E.false$ to $S.next$.

In translating the *if-then-else* statement $S \rightarrow$ if E then S_1 else S_2 , the code for the Boolean expression E has jumps out of it to the first instruction of the code for S_1 if E is true, and to the first instruction of the code for S_2 if E is false, as shown in fig. 4.11 (b).

The code for $S \rightarrow$ while E do S_1 is formed as shown in fig. 4.11 (c). A new label $S.begin$ is created and attached to the first instruction generated for E . Another new label $E.true$ is attached to the first instruction for S_1 .

Q.18. Describe the control-flow translation of Boolean expressions.

Ans. For the $E.code$, the code produced for the Boolean expressions E in fig. 4.12. As we have indicated, E is translated into a sequence of three-address statements that evaluates E as a sequence of conditional and unconditional jumps to one of two locations – $E.true$, the place control is to reach if E is true and $E.false$, the place control is to reach if E is false. The basic idea behind the translation is the following –

Suppose E is of the form $a < b$, then the generated code is of the form

if $a < b$ goto $E.true$
 goto $E.false$

Suppose E is of the form E_1 or E_2 . If E_1 is true, then we immediately know that E itself is true, so $E_1.true$ is the same as $E.true$. If E_1 is false, then E_2 must be evaluated, so we make $E_1.false$ be the label of the first statement in the code for E_2 . The true and false exits of E_2 can be made the same as the true and false exits of E , respectively.

Production	Semantic Rules
$S \rightarrow \text{if } E \text{ then } S_1$	$E.true := \text{newlabel};$ $E.false := S.next;$ $S_1.next := S.next;$ $S.code := E.code $ $\quad \text{gen}(E.true ':') S_1.code$
$S \rightarrow \text{if } E \text{ then } S_1 \text{ else } S_2$	$E.true := \text{newlabel};$ $E.false := \text{newlabel};$ $S_1.next := S.next;$ $S_2.next := S.next;$ $S.code := E.code $ $\quad \text{gen}(E.true ':') S_1.code $ $\quad \text{gen('goto' S.next)} $ $\quad \text{gen}(E.false ':') S_2.code$
$S \rightarrow \text{while } E \text{ do } S_1$	$S.begin := \text{newlabel};$ $E.true := \text{newlabel};$ $E.false := S.next;$ $S_1.next := S.begin;$ $S.code := \text{gen}(S.begin ':') E.code $ $\quad \text{gen}(E.true ':') S_1.code $ $\quad \text{gen('goto' S.begin)}$

Fig. 4.12 Syntax-directed Definition for Flow-of-control Statements

Analogous considerations apply to the translation of E_1 and E_2 . No code is needed for an expression E of the form **not** E_1 – we just interchange the true and false exits of E_1 to get the true and false exits of E . A syntax-directed definition that generates three address code for Boolean expressions in this manner is shown in fig. 4.13. Note that the *true* and *false* attributes are inherited.

Production	Semantic Rules
$E \rightarrow E_1 \text{ or } E_2$	$E_1.true := E.true;$ $E_1.false := \text{newlabel};$ $E_2.true := E.true;$ $E_2.false := E.false;$ $E.code := E_1.code \text{gen}(E_1.false ':') E_2.code$
$E \rightarrow E_1 \text{ and } E_2$	$E_1.true := \text{newlabel};$ $E_1.false := E.false;$ $E_2.true := E.true;$ $E_2.false := E.false;$ $E.code := E_1.code \text{gen}(E_1.true ':') E_2.code$
$E \rightarrow \text{not } E_1$	$E_1.true := E.false;$ $E_1.false := E.true;$ $E.code := E_1.code$
$E \rightarrow (E_1)$	$E_1.true := E.true;$ $E_1.false := E.false;$ $E.code := E_1.code$
$E \rightarrow id_1 \text{ relop } id_2$	$E.code := \text{gen}('if' id_1.place \text{ relop.op } id_2.place 'goto' E.true) $ $\quad \text{gen('goto' E.false)}$
$E \rightarrow \text{true}$	$E.code := \text{gen('goto' E.true)}$
$E \rightarrow \text{false}$	$E.code := \text{gen('goto' E.false)}$

Fig. 4.13 Syntax-directed Definition to Produce Three Address Code for Booleans

Q.19. Write a top down translation scheme to produce quadruples for Boolean expression.
 (R.G.P.V., Dec. 2003, June 2007)

Ans. In programming languages, Boolean expressions are used to compute logical values. But more often these are used as conditional expressions in statements. Such as **if-then**, **if-then-else**, or **while-do** statements.

Boolean expressions are composed of the Boolean operators (AND, OR, and NOT) applied to elements that are Boolean variables or relational expressions. Relational expression are of the form $E_1 \text{ relop } E_2$, where E_1 and E_2 are arithmetic expressions.

There are two methods of translating a Boolean expression. The first is to encode true or false numerically and 1 is used to denote true and 0 to denote false. All non-negative quantity denote true and any negative number denote false.

The second method of implementing Boolean expression is “flow of control”. This method is convenient in implementing the Boolean expression in flow-of control statements, such as **if-then** and **while-do** statements.

Implementation of Boolean Expression using Numerical Representation –

The implementation of Boolean expression using 1 to denote true and 0 to denote false. Expressions will be evaluated from left to right.

A relational expression such as $a < b$ is equivalent to the conditional statement **if** $a < b$ **then** 1 **else** 0.

A translation scheme for producing three address code for Boolean expressions is shown in fig. 4.14.

$E \rightarrow E_1 \text{ or } E_2$	$\{ E.place := \text{newtemp};$ $\quad \text{emit}(E.place ':=' E_1.place \text{ 'or' } E_2.place)$
$E \rightarrow E_1 \text{ and } E_2$	$\{ E.place := \text{newtemp};$ $\quad \text{emit}(E.place ':=' E_1.place \text{ 'and' } E_2.place)$
$E \rightarrow \text{not } E_1$	$\{ E.place := \text{newtemp};$ $\quad \text{emit}(E.place ':=' \text{ 'not' } E_1.place)$
$E \rightarrow (E_1)$	$\{ E.place := E_1.place \}$
$E \rightarrow id_1 \text{ relop } id_2$	$\{ E.place := \text{newtemp};$ $\quad \text{emit}('if' id_1.place \text{ relop.op } id_2.place 'goto' \text{ nextstat}+3);$ $\quad \text{emit}(E.place ':=' '0');$ $\quad \text{emit}('goto' \text{ nextstat} + 2);$ $\quad \text{emit}(E.place ':=' '1') \}$
$E \rightarrow \text{true}$	$\{ E.place := \text{newtemp};$ $\quad \text{emit}(E.place ':=' '1') \}$
$E \rightarrow \text{false}$	$\{ E.place := \text{newtemp};$ $\quad \text{emit}(E.place ':=' '0') \}$

Fig. 4.14 Translation Scheme using a Numerical Representation for Booleans

The three address code for the expression $a < b$ or $c < d$ and $e < f$ is shown in fig. 4.15.

```

100 : if a < b goto 103
101 : t1 := 0
102 : goto 104
103 : t1 := 1
104 : if c < d goto 107
105 : t2 := 0
106 : goto 108
107 : t2 := 1
108 : if e < f goto 111
109 : t3 := 0
110 : goto 112
111 : t3 := 1
112 : t4 := t2 and t3
113 : t5 := or t4

```

Fig. 4.15 Translation of $a < b$ or $c < d$ and $e < f$

Above three address code can be implemented in record structure like quadruples. Quadruples is a record structure with four fields like op, arg1, arg2 and result.

Q.20. Write a short note on case statements.

Ans. The “switch” or “case” statement is available in a variety of languages; even the Fortran computed and assigned goto’s can be regarded as varieties of the switch statement. Out switch-statement syntax is shown in fig. 4.16.

There is a selector expression, which is to be evaluated, followed by n constant values that the expression might take, perhaps including a default “value”, which always matches the expression if no other value does. The intended translation of a switch code is to –

```

switch expression
begin
  case value: statement
  case value: statement
  ...
  case value: statement
  default: statement
end

```

Fig. 4.16 Switch-statement Syntax

- (i) Evaluate the expression.
- (ii) Find which value in the list of cases is the same as the value of the expression. The default value matches the expression if none of the values explicitly mentioned in cases does.
- (iii) Execute the statement associated with the value found.

Step (ii) is an n-way branch, which can be implemented in one of several ways. If the number of cases is not too great, say 10 at most, then it is reasonable to use a sequence of conditional goto’s, each of which tests for an individual value and transfers to the code for the corresponding statement.

A more compact way to implement this sequence of conditional goto’s is to create a table of pairs, each pair consisting of a value and a label for the code of the corresponding statement. Code is generated to place at the end of this table the value of the expression itself, paired with the label for the default statement. A simple loop can be generated by the compiler to compare the value of the expression with each value in the table, being assured that if no other match is found, the last entry is sure to match.

If the number of values exceeds 10 or so, it is more efficient to construct a hash table for the values, with the labels of the various statements as entries. If no entry for the value possessed by the switch expression is found, a jump to the default statement can be generated.

Q.21. Discuss about the syntax directed translation of case statements.
Or

Write a syntax-directed definition to translate ‘switch’ statement. With a suitable example, show the translation of the source language ‘switch’ statement. (R.G.P.V., June 2010)

Ans. Consider the following switch statements

```

switch E
begin
  case  $V_1$  :  $S_1$ 
  case  $V_2$  :  $S_2$ 
  .....
  .....
  case  $V_{n-1}$  :  $S_{n-1}$ 
  default :  $S_n$ 
end

```

with a syntax-directed translation scheme, it is convenient to translate this case statement into intermediate code that has the form of fig. 4.17.

```

code to evaluate E into t
goto test
L1 : code for  $S_1$ 
      goto next
L2 : code for  $S_2$ 
      goto next
...
Ln-1 : code for  $S_{n-1}$ 
        goto next
Ln : code for  $S_n$ 
        goto next
test : if t =  $V_1$  goto L1
        if t =  $V_2$  goto L2
        ...
        if t =  $V_{n-1}$  goto Ln-1
        goto Ln
next :

```

Fig. 4.17 Translation of a Case Statement

The tests all appear at the end so that a simple code generator can recognize the multiway branch and generate efficient code for it, using the most appropriate implementation. If we generate more straightforward sequence shown in fig. 4.18, the compiler would have to do extensive analysis to find the most efficient implementation. To translate into the form of fig. 4.17, when we see the keyword **switch**, we generate two new labels *test* and *next*, and a new temporary *t*. Then as we parse the expression *E*, we generate code to evaluate *E* into *t*. After processing *E*, we generate the jump *goto test*.

```

code to evaluate E into t
if t ≠ V1 goto L1
code for S1
goto next
L1 : if t ≠ V2 goto L2
code for S2
goto next
L2 : ...
Ln-2 : if t ≠ Vn-1 goto Ln-1
code for Sn-1
goto next
Ln-1 : code for Sn
next :

```

Fig. 4.18 Another Translation of Case Statement

Then as we see each **case** keyword, we create a new label *L_i* and enter it into the symbol table. We place on a stack, used only to store cases, a pointer to this symbol-table entry and the value *V_i* of the case constant.

We process each statement **case** *V_i* : *S_i* by emitting the newly created label *L_i*, followed by the code for *S_i*, followed by the jump *goto next*. Then when the keyword **end** terminating the body of the switch is found, we are ready to generate the code for the *n*-way branch. Reading the pointer-value pairs on the case stack from the bottom to the top, we can generate a sequence of three address statements of the form

```

case V1 L1
case V2 L2
.....
.....
case Vn-1 Ln-1
case t Ln
label next

```

where *t* is the name holding the value of the selector expression *E*, and *L_n* is the label for the default statement. The case *V_i* *L_i* three address statement is a synonym for *if t = V_i goto L_i* in fig. 4.17, but for the case it is easier for the final code generator to detect as a candidate for special treatment.

For example, consider the following switch statement –

```

switch (i + j)
{
    case1 : x = y + z
    default : p = q + r
    case2 : u = v + w
}

```

The above translation scheme translates into the following three address code, which is also shown in fig. 4.19.



Fig. 4.19 Contents of Queue During the Translation

```

i)      t1 = i + j
i + 1)  goto(i + t1)
i + 2)  t1 = y + z
i + 3)  x = t1
i + 4)  goto _
i + 5)  t2 = q + r
i + 6)  p = t2
i + 7)  goto _
i + 8)  t3 = v + w
i + 9)  u = t3
i + 10) goto _
i + 11) if t1 = 1 goto(i+2)
i + 12) if t1 = 2 goto(i+8)
i + 13) goto(i + 5)

```

Q.22. Write short note on backpatching.

(R.G.P.V., June 2005, 2006, Dec. 2016, 2017, Nov. 2018)

Or

Describe the backpatching technique.

(R.G.P.V., Dec. 2015)

Or

What do you mean by backpatching ?

(R.G.P.V., June 2016)

Ans. Grouping some of the phases into one pass is not that easy. Grouping intermediate and object code-generation phases is difficult, because it is often very hard to perform object code generation until a sufficient number of intermediate code statements have been generated. Here, the interface between the two is not based on only one intermediate instruction certain languages permit the use of a variable before it is declared. Similarly, many languages also permit forward jumps. Therefore, it is not possible to generate object

code for a construct sufficient intermediate code statements have been generated. To overcome this problem and enable the merging of intermediate and object code generation into one pass, the technique called **backpatching** is used; the object code is generated by leaving 'statement holes', which will be filled later when the information becomes available.

For specificity, we generate quadruples into a quadruple array. Labels will be indices into this array. To manipulate lists of labels, we use three functions –

- (i) **makelist(i)** creates a new list containing only i, an index into the array of quadruples; **make list** returns a pointer to the list it has made.
- (ii) **Merge(P₁, P₂)** concatenates the lists pointed to by P₁ and P₂ and returns a pointer to the concatenated list.
- (iii) **Backpatch(P, i)** inserts i as the target label for each of the statements on the list pointed to by P.

Q.23. Define backpatching and semantic rules for Boolean expression. How DAG is different from syntax tree ?
(R.G.P.V., May 2018)

Ans. Refer to Q.22.

A translation scheme is created which is appropriate for producing quadruples for Boolean expressions during bottom-up parsing. A marker nonterminal M is inserted into the grammar to cause a semantic action to pickup, at appropriate times, the index of the next quadruple to be produced. The grammar used is as follows –

- (i) $E \rightarrow E_1 \text{ or } M E_2$
- (ii) $| E_1 \text{ and } M E_2$
- (iii) $| \text{ not } E_1$
- (iv) $| (E_1)$
- (v) $| id_1 \text{ relop } id_2$
- (vi) $| \text{ true}$
- (vii) $| \text{ false}$
- (viii) $M \rightarrow \epsilon$

To generate jumping code for Boolean expressions, synthesized attributes truelist and falselist of nonterminal E are employed. When code is generated for E, jumps to the true and false exits are left incomplete, with the label field unfilled. These incomplete jumps are placed on lists pointed to by E.truelist and E.falselist, as suitable.

The semantic actions reflect the above considerations. Consider the production $E \rightarrow E_1 \text{ and } M E_2$. When E₁ is false, then E is also false, so the statements on E₁.falselist become part of E.falselist. But, when E₁ is true, next E₂ must be examined, so the target for the statements E₁.truelist must be the starting of the code generated for E₂. The marker nonterminal M is used to obtain this target. Attribute M.quad records the number of the first statement of E₂.code. With the production $M \rightarrow \epsilon$, the semantic action associated is,

{M.quad := nextquad}

The variable nextquad holds the index of the next quadruple to follow. This value will be backpatched onto the E₁.truelist if remainder of production $E \rightarrow E_1 \text{ and } M E_2$ is seen. The translation scheme is given below –

- (i) $E \rightarrow E_1 \text{ or } M E_2$ {backpatch(E₁.falselist, M.quad);
E.truelist := merge(E₁.truelist, E₂.truelist);
E.falselist := E₂.falselist}
- (ii) $E \rightarrow E_1 \text{ and } M E_2$ {backpatch(E₁.truelist, M.quad);
E.truelist := E₂.truelist;
E.falselist := merge(E₁.falselist, E₂.falselist)}
- (iii) $E \rightarrow \text{not } E_1$ {E.truelist := E₁.falselist;
E.falselist := E₁.truelist}
- (iv) $E \rightarrow (E_1)$ {E.truelist := E₁.truelist;
E.falselist := E₁.falselist}
- (v) $E \rightarrow id_1 \text{ relop } id_2$ {E.truelist := makelist(nextquad);
E.falselist := makelist(nextquad + 1);
emit('if' id₁.place relop.op id₂.place 'goto_')
emit('goto_')}
- (vi) $E \rightarrow \text{true}$ {E.truelist := makelist(nextquad);
emit('goto_')}
- (vii) $E \rightarrow \text{false}$ {E.falselist := makelist(nextquad);
emit('goto_')}
- (viii) $M \rightarrow \epsilon$ {M.quad := nextquad}

For simplicity, semantic action (v) produces two statements, a conditional goto and an unconditional one. Neither has its target filled in. The index of the first generated statement is made into a list, and E.truelist is given a pointer to that list. The second generated statement goto_ is also made into a list and given to E.falselist.

The difference between DAG and syntax tree is that in DAG the common subexpressions include more than one parent while in syntax tree the common subexpression would be shown as duplicated subtree.

Q.24. Define backpatching and semantic rules for Boolean expression.
(R.G.P.V., May 2019)

Ans. Refer to Q.23.

Q.25. What are procedure calls ?

Or

What are the procedure calling and returning sequences ? Explain the sequence of actions in each of them.
(R.G.P.V., Dec. 2015)

Ans. The procedure is an important and frequently used programming construct. It is imperative for a compiler to generate good code for procedure calls and returns. The run-time routines that handle procedure argument passing, calls and returns are part of the run-time support package.

Let us consider a grammar for a simple procedure call statement –

- (i) $S \rightarrow \text{call id (Elist)}$
- (ii) $\text{Elist} \rightarrow \text{Elist}, E$
- (iii) $\text{Elist} \rightarrow E$

When a procedure call occurs, space must be allocated for the activation record of the called procedure. The arguments of the called procedure must be evaluated and made available to the called procedure in a known place. Environment pointers must be established to enable the called procedure to access data in enclosing blocks. The state of the calling procedure must be saved so it can resume execution after the call. Also saved in a known place is the return address, the location to which the called routine must transfer after it is finished. The return address is usually the location of the instruction that follows the call in the calling procedure. Finally, a jump to the beginning of the code for the called procedure must be generated.

There is no exact division of the run-time tasks between the calling and called procedure. Often the source language, the target machine, and the operating system impose requirements that favor one solution over another.

Q.26. How is a call to a procedure translated into TAC? Illustrate with an example. (R.G.P.V., May 2019)

Ans. Refer to Q.25.

For Example – Consider a grammar for a simple procedure call,

- $S \rightarrow \text{Call id (L)}$
- $L \rightarrow L, E$
- $L \rightarrow E$

Here S denotes the statement and L denotes the list of parameters and E denotes the expression.

The translation scheme can be as given in table 4.1 –

Table 4.1 Syntax Directed Translation Scheme to Generate Three Address Code (TAC) for Procedure Call

Production Rule	Semantic Action
$S \rightarrow \text{Call id (L)}$	{for each item p in queue do append ('param' p); append ('call' id.place);}
$L \rightarrow L, E$	{insert E.place in the queue}
$L \rightarrow E$	{initialize the queue and insert E.place in the queue}

The data structure queue is used to hold the various parameters of the procedure. The keyword param is used to denote list of parameters passed to the procedure. The call to the procedure is given by 'call id' where id denotes the

name of procedure, and E.place gives the value of parameter which is inserted in the queue.

For $L \rightarrow E$, the queue gets empty and a single pointer to the symbol table is obtained. This pointer denotes the value of E.

NUMERICAL PROBLEMS

Prob.1. Give three address code for the following code fragment –

```

If a < b then
    while c > d do
        x = x - y
    else
        do
            p = p + q
            while e <= f

```

(R.G.P.V., June 2010)

Sol. (1) If (a < b) goto (3) (2) goto (8)
 (3) If (c > d) goto (5) (4) goto (11)
 (5) $t_1 = x - y$ (6) $x := t_1$
 (7) goto (11) (8) $t_2 = p + q$
 (9) $p := t_2$ (10) if (e <= f) goto (8)
 (11)

Prob.2. Construct 3-address code for the following –

```

If [(a < b) and ((c > d) or (a > d))], then
    z = x + y * z
else
    z = z + 1

```

(R.G.P.V., Dec. 2008, 2010, 2011)

Sol. The three address code for the given program is as follows –

- (1) if a < b goto (3)
- (2) goto (11)
- (3) if c > d goto (7)
- (4) goto (5)
- (5) if a > d goto (7)
- (6) goto (11)
- (7) $t_1 = y * z$
- (8) $t_2 = x + t_1$
- (9) $z = t_2$
- (10) goto (13)
- (11) $t_3 = z + 1$
- (12) $z = t_3$
- (13)

Prob.3. Generate the three address code for the following program fragment –

while ($A < C$ and $B > D$) do

if $A = 1$ then

$C := C + 1$

else

while $A \leq D$ do

$A := A + 3$

(R.G.P.V., Dec. 2006, 2009, June 2011)

Sol. The three address code for the given program fragment is as follows –

- (1) if $A < C$ goto (3)
- (2) goto (15)
- (3) if $B > D$ goto (5)
- (4) goto (15)
- (5) if $A = 1$ go to (7)
- (6) goto (15)
- (7) $t_1 := C + 1$
- (8) $C := t_1$
- (9) goto (1)
- (10) if $A \leq D$ goto (12)
- (11) goto (1)
- (12) $t_2 := A + 3$
- (13) $A := t_2$
- (14) goto (10)
- (15)

Prob.4. Switch the backpatching. Also generate three address code for the following program fragment –

while ($A < B$ and $C > D$) do

if $A = 1$ then

$A := A + 1$

$A := A + 3$

(R.G.P.V., June 2012)

Sol. Backpatching – Refer to Q.22.

Three Address Code – Refer to Prob.3.

Prob.5. Write down the three address code for the following switch statement –

Switch ($i + j$)

{

case 1 : $x := y - z$

case 2 : $a := b + c$

case 3 : $i = j + k$

}

(R.G.P.V., Dec. 2013)

Sol. The three address code for the given program fragment is as follows –

- (1) $t_1 = i + j$
- (2) goto (12)
- (3) $t_1 = y - z$
- (4) $x = t_1$
- (5) goto (15)
- (6) $t_2 = b + c$
- (7) $a = t_2$
- (8) goto (15)
- (9) $t_3 = j + k$
- (10) $i = t_3$
- (11) goto (15)
- (12) if $t_1 = 1$ goto (3)
- (13) if $t_1 = 2$ goto (6)
- (14) if $t_1 = 3$ goto (9)
- (15)

Prob.6. Translate the following code into a three address code –

$a = 10;$

$b = 6;$

if ($a > b + 5$)

$b = 5;$

(R.G.P.V., Dec. 2017)

Sol. The three address code for the given code is as follows –

- (1) $a = 10$
- (2) $b = 6$
- (3) $t_1 = b + 5$
- (4) if ($a > t_1$) goto (5)
- (5) $b = 5$

Prob.7. Consider the following while-statement –

while $A > B$ and $A \leq 2*B - 5$ do

$A := A + B;$

(i) Construct the parse tree for the given above while-statement

(ii) Write the intermediate code for while-statement.

(R.G.P.V., Dec. 2003, 2007)

Sol. (i) Parse Tree – The parse tree for this statement is shown in fig. 4.20. We use “exp” for expression, “relop” for relational operator and we indicate parenthetically the particular name or constant to which each instance of token **id** and **const** refer.

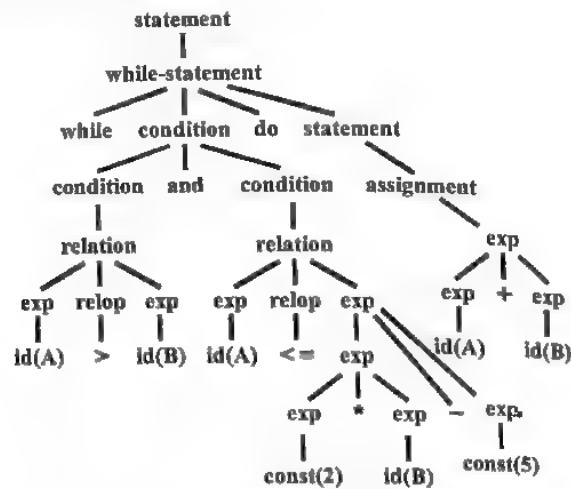


Fig. 4.20 Parse Tree for while-statement

(ii) Intermediate Code –

L_1 : if $A > B$ goto L_2

goto L_3

L_2 : $T_1 := 2 * B$

$T_2 := T_1 - 5$

if $A \leq T_2$ goto L_4

goto L_3

L_4 : $A := A + B$

goto L_1

L_3 : END

Fig. 4.21 Intermediate Code for while-statement

Prob.8. Translate the following statement to quadruple –

if $a > b$ then $x = a + b$

else $x = a - b$

(R.G.P.V., June 2009)

Sol. The three address code for the given statement is –

(1) if $(a > b)$ goto (3) (2) goto (6)

(3) $t_1 := a + b$ (4) $x := t_1$

(5) goto (8) (6) $t_2 := a - b$

(7) $x := t_2$ (8)

Quadruple – The quadruple of above expression is as follows –

	Operator	Operand1	Operand2	Result
(1)	>	a	b	(3)
(2)				(6)
(3)	+	a	b	t_1
(4)	=	t_1		x
(5)				(8)
(6)	-	a	b	t_2
(7)	=	t_2		x
(8)				

Prob.9. Write quadruples and triples for given expression –

$Z = a - b * c \uparrow d / e + f$

(R.G.P.V., June 2016)

Sol. The three address code of the given expression is as follows –

$t_1 := c \uparrow d$

$t_2 := b * t_1$

$t_3 := t_2 / e$

$t_4 := a - t_3$

$t_5 := t_4 + f$

$Z := t_5$

The quadruples of three address code are shown below –

	Operator	Operand1	Operand2	Result
(1)	\uparrow	c	d	t_1
(2)	*	b	t_1	t_2
(3)	/	t_2	e	t_3
(4)	-	a	t_3	t_4
(5)	+	t_4	f	t_5
(6)	:=	t_5		Z

The triples of three address code are shown below –

	Operator	Operand1	Operand2
(1)	\uparrow	c	d
(2)	*	b	(1)
(3)	/	(2)	e
(4)	-	a	(3)
(5)	+	(4)	f
(6)	:=	Z	(5)

Prob.10. Translate the expression –

$$A := -B * (c + d)/E$$

into quadruples and triples representations.

(R.G.P.V., June 2012)

Sol. The three address code of the given expression is as follows –

$$\begin{aligned} t_1 &:= -B \\ t_2 &:= c + d \\ t_3 &:= t_1 * t_2 \\ t_4 &:= t_3 / E \\ A &:= t_4 \end{aligned}$$

Quadruples – The quadruples of three address statements are represented as –

	Operator	Operand1	Operand2	Result
(1)	–	B		t_1
(2)	+	c	d	t_2
(3)	*	t_1	t_2	t_3
(4)	/	t_3	E	t_4
(5)	:=	t_4		A

Triples – The triples of three address statements are represented as –

	Operator	Operand1	Operand2
(1)	–	B	
(2)	+	c	d
(3)	*	(1)	(2)
(4)	/	(3)	E
(5)	:=	A	(4)

Prob.11. Write quadruples from the expression –

$$(a + b) * (c + d) - (a + b + c)$$

(R.G.P.V., Dec. 2014)

Sol. The three address code of the given expression is as follows –

$$\begin{aligned} t_1 &= a + b \\ t_2 &= c + d \\ t_3 &= t_1 * t_2 \\ t_4 &= a + b \\ t_5 &= t_4 + c \\ t_6 &= t_3 - t_5 \end{aligned}$$

	Operator	Operand1	Operand2	Result
(1)	+	a	b	t_1
(2)	+	c	d	t_2
(3)	*	t_1	t_2	t_3
(4)	+	a	b	t_4
(5)	+	t_4	c	t_5
(6)	–	t_3	t_5	t_6

Prob.12. Write triples for the expression

$$(a + b) * (c + d) - (a + b + c)$$

(R.G.P.V., Dec. 2012)

Sol. The three address code of the given expression is as follows –

$$\begin{aligned} t_1 &= a + b \\ t_2 &= c + d \\ t_3 &= t_1 * t_2 \\ t_4 &= a + b \\ t_5 &= t_4 + c \\ t_6 &= t_3 - t_5 \end{aligned}$$

	Operator	Operand1	Operand2
(1)	+	a	b
(2)	+	c	d
(3)	*	(1)	(2)
(4)	+	a	b
(5)	+	(4)	c
(6)	–	(3)	(5)

into quadruples, triples and indirect triples.

(R.G.P.V., Dec. 2010)

Or

Write quadruples, triples and indirect triples for the expression –

$$-(a + b) * (c + d) - (a + b + c)$$

(R.G.P.V., Dec. 2003, June 2004, 2007, 2008, Dec. 2009, 2011)

Or

Translate the following expression to quadruple, triple and indirect triple

$$-(x + y) * (z + c) - (x + y + z)$$

(R.G.P.V., June 2009)

Sol. The three address code of the given expression is as follows –

$t_1 := a + b$
 $t_2 := -t_1$
 $t_3 := c + d$
 $t_4 := t_2 * t_3$
 $t_5 := a + b$
 $t_6 := t_5 + c$
 $t_7 := t_4 - t_6$

(i) **Quadruples** – The quadruples of the given expression are shown below –

	Operator	Operand1	Operand2	Result
(1)	+	a	b	t_1
(2)	-	t_1		t_2
(3)	+	c	d	t_3
(4)	*	t_2	t_3	t_4
(5)	+	a	b	t_5
(6)	+	t_5	c	t_6
(7)	-	t_4	t_6	t_7

(ii) **Triples** – The triples of the given expression are shown below –

	Operator	Operand1	Operand2
(1)	+	a	b
(2)	-	(1)	
(3)	+	c	d
(4)	*	(2)	(3)
(5)	+	a	b
(6)	+	(5)	c
(7)	-	(4)	(6)

(iii) **Indirect Triples** – The indirect triples of the given expression are shown below –

	STATEMENT		Operator	Operand1	Operand2
(1)	(10)	(10)	+	a	b
(2)	(20)	(20)	-	(10)	
(3)	(30)	(30)	+	c	d
(4)	(40)	(40)	*	(20)	(30)
(5)	(50)	(50)	+	a	b
(6)	(60)	(60)	+	(50)	c
(7)	(70)	(70)	-	(40)	(60)

Prob.14. Translate the following expression to quadruple and triple –
 $-(X - Y)/(Z * C) - (X + Y - Z)$

(R.G.P.V., Dec. 2015)

Sol. The three address code of the given expression is as follows –

$t_1 = X - Y$
 $t_2 = -t_1$
 $t_3 = Z * C$
 $t_4 = t_2/t_3$
 $t_5 = X + Y$
 $t_6 = t_5 - Z$
 $t_7 = t_4 - t_6$

The quadruples of the given expression are shown below –

	Operator	Operand1	Operand2	Result
(1)	-	X	Y	t_1
(2)	-	t_1		t_2
(3)	*	Z	C	t_3
(4)	/	t_2	t_3	t_4
(5)	+	X	Y	t_5
(6)	-	t_5	Z	t_6
(7)	-	t_4	t_6	t_7

The triples of given expression are shown below –

	Operator	Operand1	Operand2
(1)	-	X	Y
(2)	-	(1)	
(3)	*	Z	C
(4)	/	(2)	(3)
(5)	+	X	Y
(6)	-	(5)	Z
(7)	-	(4)	(6)

Prob.15. Using backpatching, generate an intermediate code for the following expression –

$$A < B \text{ OR } C < D \text{ AND } P < Q$$

(R.G.P.V., Dec. 2011)

Sol. The parse tree for the given expression using the usual order of precedence is shown in fig. 4.22.

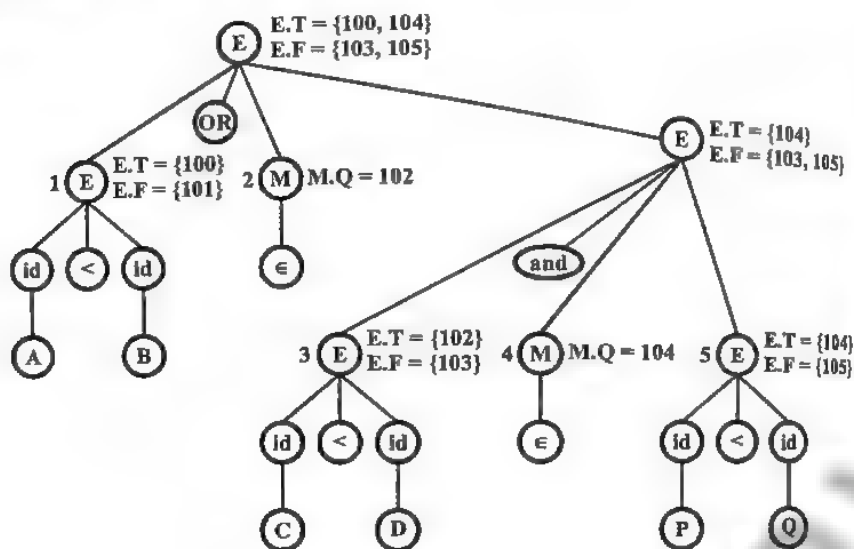


Fig. 4.22 Parse Tree

Suppose that NEXTQUAD has the initial value 100 and is incremented with each call to GEN. We have also shown the values of the translations at each node. TRUE, FALSE and QUAD are indicated by T, F and Q respectively.

Let the semantic actions occurring as the numbered nodes are created in a bottom-up manner. The two quadruples

100 : if A < B goto _

101 : goto _

are generated in order to the reduction corresponding to node 1. Node 2, then records the value of NEXTQUAD, i.e. 102. Node 3 generates the quadruples

102 : if C < D goto _

103 : goto _

Now, the current value of NEXTQUAD which is 104 is recorded by node 4. Node 5 generates the quadruples

104 : if P < Q goto _

105 : goto _

Node 6 corresponds to a reduction by $E \rightarrow E^{(1)}$ and $ME^{(2)}$. The corresponding semantic routine calls BACKPATCH ({102}, 104) where {102} as argument denotes a pointer to the list containing only 102, that list being the one pointed to by E-TRUE of node 3. This calls to BACKPATCH fills in 104 in quadruple 102. The six quadruples generated so far are –

100 : if A < B goto _

101 : goto _

102 : if C < D goto _

103 : goto _

104 : if P < Q goto _

105 : goto _

The root, node 7, is created by a reduction by $E \rightarrow E^{(1)}$ or $ME^{(2)}$. The associated semantic routine calls BACKPATCH ({101}, 102) which leaves the quadruples like –

100 : if A < B goto _

101 : goto 102

102 : if C < D goto 104

103 : goto _

104 : if P < Q goto _

105 : goto _

CODE GENERATION – ISSUES IN THE DESIGN OF CODE GENERATOR, BASIC BLOCKS AND FLOW GRAPHS, REGISTER ALLOCATION AND ASSIGNMENT

Q.27. Write short note on code generation. (R.G.P.V., Dec. 2002)

Ans. Code generation is the last phase in the compilation process. It takes as input an intermediate representation of the source program and produces as

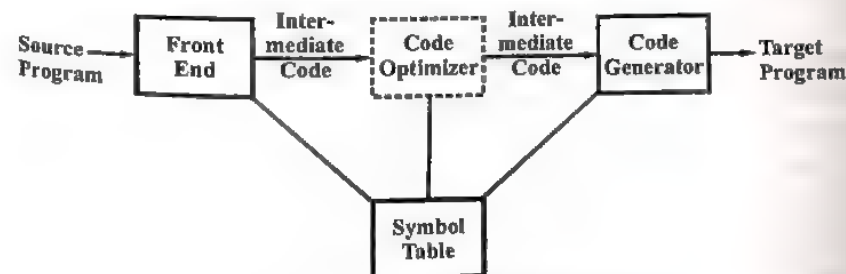


Fig. 4.23 Position of Code Generator

output an equivalent target program, as shown in fig. 4.23. Such a phase tries to transform the intermediate code into a form from which more efficient target code can be produced.

Being a machine dependent phase, it is not possible to generate good code without considering the details of the particular machine for which the compiler is expected to generate code. The requirement traditionally imposed on a code generator are severe. The output code must be correct and of high quality, meaning that it should make effective use of the resources of the target machine. Moreover, the code generator itself should run efficiently. A carefully designed code generation algorithm can easily produce code that is several times faster than that produced by hastily conceived algorithm.

Q.28. What are the problems encountered in code generation ?

(R.G.P.V., Dec. 2007, June 2012)

Or

Discuss the factors affecting target code generation. (R.G.P.V., Dec. 2012)

Ans. There are three main difficulties that we face when attempting to generate efficient object code, namely –

(i) **Selection of the most-efficient instructions to represent the computation specified by the three-address statement** – Many machines allow certain computations to be done in more than one way. For example, if a machine permits an instruction AOS for incrementing the contents of a storage location directly, then for a three-address statement $a = a + 1$, it is possible to generate the instruction AOS, a, rather than a sequence of instructions like the following –

MOVE a, R

ADD#1, R

MOVE R, a

Deciding which instruction sequence is better in the problem requires an extensive knowledge about the context in which these three-address statements will appear.

(ii) **Deciding on the computation order that will lead to the generation of more-efficient object code** – Some computation orders require fewer registers to hold intermediate results than other. Now, deciding the best order is very difficult. For example, consider the basic block –

$t_1 = a + b$

$t_2 = c + d$

$t_3 = e - t_2$

$t_4 = t_1 - t_3$

If the order of computation used is the one given in the basic block $t_1-t_2-t_3-t_4$, then the number of registers required for holding the intermediate result is more than when the order $t_2-t_3-t_1-t_4$ is used.

(iii) **Deciding on Registers** – Deciding which register should handle the computation is another problem that stands in the way of good code generation. The problem is further complicated when a machine requires register-pairs for some operands and results.

Q.29. What are the general issues in designing a code generator ?

(R.G.P.V., June 2008, Dec. 2009, June 2012)

Or

Discuss the issues in the design of code generator.

(R.G.P.V., Dec. 2013, 2014)

Or

Discuss some commonly used issues in code generation.

(R.G.P.V., June 2016)

Or

Explain various issues in design of code generator. (R.G.P.V., Dec. 2016)

Or

Explain in brief the various issues of design of a code generator.

(R.G.P.V., Nov. 2018)

Ans. The generic issues in the design of code generators are –

(i) **Input to the Code Generator** – The code generator input consists of the intermediate representation of the source program produced by the front end, together with information in the symbol table that is used to determine the run time addresses of the data objects denoted by the names in the intermediate representation.

Intermediate language can be represented in several ways – Linear representation such as postfix notation, three-address representation such as quadruples, virtual machine representations such as stack machine code and graphical representations such as syntax trees and DAGs.

(ii) **Target Program** – The output of the code generator is the target program. This output can be of forms absolute machine language, relocatable machine language, or assembly language.

Producing an absolute machine language program as output has the advantage that it can be placed in a fixed location in memory and immediately executed. Producing a relocatable machine language program (object module) as output allows subprograms to be compiled separately. A set of relocatable object modules can be linked together and loaded for execution by a linking loader. The added expenses of linking and loading are paid if we produce

relocatable object modules. Producing an assembly language program as output makes the process of code generation easier. The symbolic instruction is generated and the macro facilities of the assembler is used to help to generate the code. The price paid is the assembly step after code generation. Because producing assembly code does not duplicate the entire task of the assembler.

(iii) Memory Management – Mapping names in the source program to addresses of data objects in run time memory is done co-operatively by front-end and the code generator. A name in a three-address statement refers to a symbol-table entry for the name. Symbol table entries are created as the declarations in a procedure are examined. The type in a declaration determines the width i.e., the amount of storage, needed for the declared name. A relative address can be determined for the name in a data area for the procedure, from the symbol table information.

If machine code is being generated, labels in three-address statements have to be converted to address of instructions. This process is analogous to the “backpatching” technique. Suppose labels refer to quadruple numbers in a quadruple array. As each quadruple is scanned the location of first machine instruction is deduced generated for that quadruple, by maintaining a count of the number of words used for the instructions generated. This count is kept in quadruple array. If a reference such as $j: \text{goto } i$ is encountered and $i < j$, the current quadruple number, a jump instruction is generated with the target address equal to the machine location of the first instruction in the code for quadruple i . If the jump is forward, i machine instruction generated for quadruple j . Then, when we process quadruple i , we fill in the proper machine location for all instructions that are forward jumps to i .

(iv) Instruction Selection – The nature of the instruction set of the target machine determines the difficulty of instruction selection. The uniformity and completeness of the instruction set are important factors. If target machine does not support every data type in a uniform manner, then each exception to the general rule requires special handling.

Instruction speeds and machine idioms are other important factors. If efficiency of the target program is not to be taken care, instruction selection is straightforward.

For each type of three-address statement, we design a code skeleton that outlines the target code to be generated for the construction. The quality of the generated code is determined by its speed and size, so size of the generated code should be less.

(v) Register Allocation – Instruction involving register operands are shorter and faster than those involving operands in memory. The use of registers

is often subdivided into two subproblems –

(a) During register allocation, the set of variables are selected that will reside in registers at a point in the program.

(b) During a subsequent register assignment phase, the specific register is selected that a variable will reside in.

Finding an optimal register assignment to variables is difficult, even with single register values. Certain machines require register pairs for some operands and results. For example, in the IBM system/370 machines integer multiplication and integer division involve register pairs.

The multiplication instruction is of the form

M X, Y

where X, the multiplicand, is the even register of an even/odd register pair. The multiplicand value is taken from odd register of the pair. The multiplier Y is a single register. The product occupies the entire even/odd register pair.

Another example shown in fig. 4.24, where R_i stands for register i . L, ST and A stand for load, store and add, respectively.

L	R_1, a	L	R_0, a
A	R_1, b	A	R_0, b
M	R_0, c	A	R_0, c
D	R_0, d	SRDA	$R_0, 32$
ST	R_1, t	D	R_0, d
		ST	R_1, t

Fig. 4.24 Optimal Machine Code Sequence

(vi) Evaluation Order – The order in which computation is performed can affect the efficiency of the target code. Some computation order require fewer registers to hold intermediate results than others. Picking a best order is another difficult NP problem which can be avoided by generating code for three-address statements in the order in which they have been produced by the intermediate code generator.

(vii) Approaches for Code Generation – The most important criterion for a code generator is that it produces correct code. Correctness takes on special significance because of the number of special cases that a code generator might face. The correct code can be easily implemented, tested and maintained. The output of a code generator can be improved by the peephole optimization.

Q.30. Explain briefly register allocation.

(R.G.P.V., Dec. 2015)

Ans. Refer Q.29 (v).

Q.31. What are the main design issues to be considered while generating code for lexical analyzer given the minimized DFA? (R.G.P.V., May 2018)

Ans. The main design issues to be considered while generating code for lexical analyzer are as follows –

(i) In a lexical analyzer, even if the input character string does not match the regular expression, there is generally some default action which needs to be taken, like echoing the character onto the screen. This may be taken care by having a stack in which input characters are stored and then control can flow to the particular action block or the default action block depending on whether it matches a regular expression or not.

(ii) There are following two methods for generating the code of standard DFA interpretation algorithm –

(a) From a template file, generate the code.

(b) Generate the code from stored static strings in the lexical analyzer generator. This method has been used in rlex.

(iii) Usually, we have a single regular expression to convert to NFA then to DFA, and interpret it. There could be a number of rules in a lexical specification file, which need to be addressed simultaneously. Consider an input lexical specification file having three rules. After translating them to NFA individually, each of them would have a start state say nfa_s1 , nfa_s2 and nfa_s3 . A single NFA can be built out of them by creating three new NFA nodes ($n1$, $n2$, $n3$) and interconnecting these using the ϵ transitions. For the entire NFA, a new NFA start node (nfa_start) can be used as a start. The following fig. 4.25 represents this. After that, this NFA can be converted to DFA and interpreted.

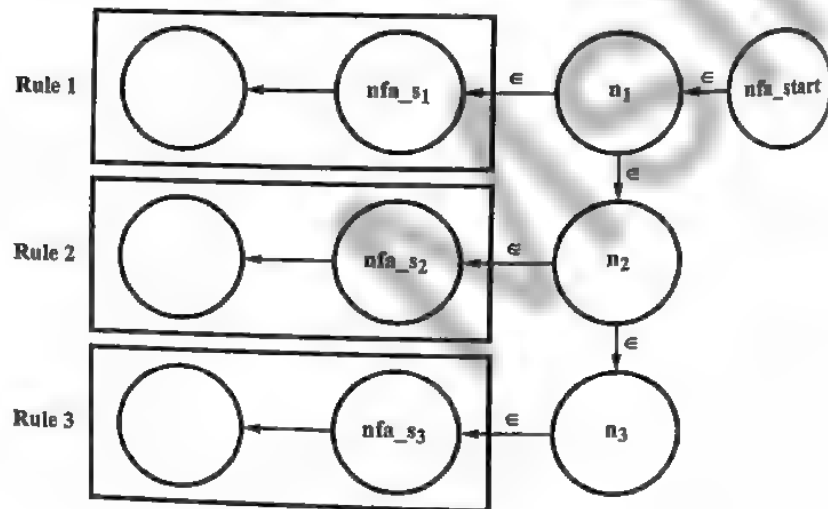


Fig. 4.25

(iv) The interpretation of DFA returns with the shortest sub-string which matches a regular expression, but a lexical analyzer requires to return the longest sub-string which matches the regular expression. This type of regular expression matching where the longest sub-string match is returned is known as greedy interpretation of the RE; for example, consider a regular expression 'abc*' and an input string 'abcccd'. In a non-greedy interpretation, the RE match would yield ab while "abcccd" would be returned by the greedy interpretation. In order to cater to greedy version the DFA Interpretation needs to be modified a little bit. The modification is to keep track of the last accepting state and continue taking the input until there is transition from accepting state to non-accepting state. If this type of transition occurs, it is a signal that the match is complete. In lexical analyzer generator rlex, code is generated for lexical analyzer that employs the greedy interpretation of DFA.

Q.32. Explain the code generation algorithm. (R.G.P.V., June 2010)

Or

Explain the code generation algorithm and generate the code for the following –

$$X = (A - B) + (A - C) + (A - C)$$

(R.G.P.V., Dec. 2013)

Ans. The code-generation algorithm takes as input a sequence of three-address statements constituting a basic block. For each three-address statement of the form $x := y \text{ op } z$ we perform the following actions –

(i) Invoke a function *getreg* to determine the location L where the result of the computation $y \text{ op } z$ should be stored. L could also be a memory location.

(ii) Consult the address descriptor for y to determine y' , the current locations of y . Prefer the register for y' if the value of y is currently both in memory and register. If the value of y is not already in L , generate the instruction $\text{MOV } y', L$ to place a copy of y in L .

(iii) Generate the instruction $\text{op } z', L$ where, z' is a current location of z . If z is in both, prefer a register to a memory location. Update the address descriptor of x to indicate that x is in location L .

(iv) If the current values of y and/or z have not next uses, are not live on exit from the block, and are in registers, alter the register descriptor to indicate that after execution of $x := y \text{ op } z$, these registers no longer will contain y and/or z , respectively.

The function *getreg* – The function *getreg* returns the location L to hold the value of x for the assignment $x := y \text{ op } z$.

For example, the assignment $x := (a - b) + (a - c) + (a - c)$ can be translated into the following three-address code sequence.

```
t := a - b
u := a - c
v := t + u
x := v + u
```

The code sequence of the above basic block is shown in fig. 4.26.

Statements	Code Generated	Register Descriptor registers empty	Address Descriptor
t := a - b	MOV a, R0 SUB b, R0	R0 contains t	t in R0
u := a - c	MOV a, R1 SUB c, R1	R0 contains t R1 contains u	t in R0 u in R1
v := t + u	ADD R1, R0	R0 contains v R1 contains u	u in R1 v in R0
x := v + u	ADD R1, R0 MOV R0, x	R0 contains x	x in R0 x in R0 and memory

Fig. 4.26 Code Sequence

Machines implement condition jumps in one of two ways. One way is to branch if the value of a designated register meets one of six conditions – negative, zero, positive, nonnegative, nonzero and nonpositive. For example,

```
if x < y goto z
```

The first call of getreg returns R0 as the location in which to compute t. Since a is not in R0, we generate instructions MOV a, R0 and SUB b, R0. We now update the register description to indicate that R0 contains t. Code generation proceeds until the last three address statement $x := u + v$ has been processed. R1 becomes empty because u has no next use. Then we generate MOV R0, x to store the live variable x at the end of the block.

Q.33. What is a basic block? With a suitable example discuss various transformation on the basic blocks. (R.G.P.V., June 2010)

Or

What is a basic block? Discuss various transformations that can be on basic block with the help of suitable example. (R.G.P.V., Dec. 2010)

Ans. A basic block is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end. The following sequence of three-

address statements forms a basic block –

```
t1 := a * a
t2 := a * b
t3 := 2 * t2
t4 := t1 + t3
t5 := b * b
t6 := t4 + t5
```

A three address statement $x := y + z$ is said to define x and to use y and z. A name in a basic block is said to be live at a given point if its value is used after that point in the program.

Transformations on Basic Blocks – A basic block computes a set of expressions. These expressions are the values of the names live on exit from the block. Two basic blocks are said to be equivalent if they compute the same set of expressions. Transformations are useful for improving the quality of code generated from a basic block. The two important classes of local transformations are –

(i) **Structure-preserving Transformation** – The primary structure-preserving transformations on basic blocks are –

(a) **Common Subexpression Elimination** – Consider the basic block

```
a := b + c
b := a - d
c := b + c
d := a - d
```

The first and third, second and fourth statements compute the same expression. This basic block may be transformed into the equivalent block

```
a := b + c
b := a - d
c := a
d := b
```

(b) **Dead Code Elimination** – The dead code means the code or variables that are never used in the basic block so these can be removed without changing the value of the basic block. For example, let x be dead, i.e. it is never used where the statement $x := y + z$ appears in a basic block then it is removed.

(c) **Renaming Temporary Variables** – A basic block can be transformed into an equivalent block in which each statement that defines a temporary defines a new temporary. Such a basic block is normal form block suppose a statement $t := b + c$, where t is a temporary. If it is changed to

$u := b + c$ where u is a new temporary variable then the value of the basic block is not changed.

(d) **Interchange of Statement** – A block with the two adjacent statements

$$t_1 := b + c$$

$$t_2 := x + y$$

Then these two statements can be interchanged without affecting the value of the block.

(ii) **Algebraic Transformations** – Countless algebraic transformations can be used to change the set of expressions computed by a basic block into an algebraically equivalent set. For example, statements such as –

$$x := x + 0$$

or $x := x * 1$

can be eliminated from a basic block without changing the set of expression it computes.

The exponentiation operator in the statement

$$x := y ** 2$$

usually requires a function call to implement. Using an algebraic transformation this can be replaced with equivalent statement

$$x := y * y$$

Q.34. What do you understand by flow graphs?

Ans. A directed graph representation of three address statement is called a flow graph. It is useful for understanding code generations algorithms. Nodes in the flow graph represent computations, and the edges represent the flow of control. The flow of control information is added to the set of basic blocks making up a program by constructing a directed graph called a flow graph. The nodes of the flow graph are the basic blocks. The initial node is the block whose leader is the first statement. There is a directed edge from block B_1 to block B_2 if B_2 can immediately follow B_1 in some execution sequence. That is, if –

(i) There is a conditional or unconditional jump from the last statement of B_1 to the first statement of B_2 .

(ii) B_2 immediately follows B_1 in the order of the program, and B_1 does not end in an unconditional jump. So, we can say that B_1 is a predecessor of B_2 and B_2 is a successor of B_1 .

Basic blocks can be represented by a variety of data structures. Each basic block can be represented by a record consisting of a count of the number of quadruples in the block, followed by a pointer to the leader of the block and by the lists of predecessors and successors of the block.

Q.35. Define “basic blocks”, flow graph with help of example.

(R.G.P.V., June 2016)

Or

Explain the basic block and control flow graph. (R.G.P.V., Nov. 2018)

Ans. Refer to Q.33 and Q.34.

Q.36. Discuss the representation of basic blocks.

Ans. Basic blocks can be represented by a variety of data structures. After partitioning the three-address statements, each basic block can be represented by a record consisting of a count of the number of quadruples in the block, followed by a pointer to the leader (first quadruple) of the block, and by the lists of predecessors and successors of the block. The quadruples are arranged or stored in each block by linked list. Quadruple numbers in jump statement at the end of basic blocks can cause problems, quadruples are moved during code optimization. A flow graph for a program is shown in fig. 4.27.

In fig. 4.27 the block B_2 running from statements (3) through (12) is the intermediate code. The (3) in *if $i \leq 20$ goto (3)* would have to be changed. Thus, we prefer to make jumps point to blocks rather than quadruples.

In a flow graph, there is one loop, consisting of block B_2 . If we define a loop, it is sufficient to say that a loop is a collection of nodes in a flow graph such that

(i) All nodes in the collection are strongly connected, that is from any node in the loop to any other, there is a path of length one or more within the loop.

(ii) The collection of nodes has a unique entry, i.e., a node in the loop such that the only way to reach a node of the loop from a node outside the loop. A loop that contains no other loops is called an inner loop.

Q.37. Write short note on register allocation and assignment.

(R.G.P.V., Dec. 2016)

Or

How CPU registers are allocated while creating machine code?

(R.G.P.V., Dec. 2014)

Ans. Instructions involving only register operands are shorter and faster than those involving memory operands. Therefore, efficient utilization of

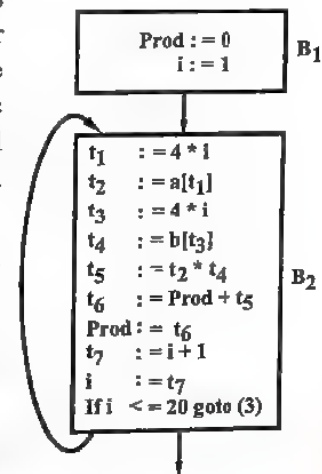


Fig. 4.27

registers is important in generating good code. Various strategies are available for deciding what values in a program should reside in registers (register allocation) and in which register each value should reside (register assignment).

One approach to register allocation and assignment is to assign specific values in an object program to certain registers. For example, a decision can be made to assign base addresses to one group of registers, arithmetic computation to another, the top of the run-time stack to a fixed integer, and so on. This approach has the advantage that it simplifies the design of a compiler. Its disadvantage is that, applied too strictly, it uses registers inefficiently; certain registers may go unused over substantial portions of code, while unnecessary loads and stores are generated.

Global Register Allocation – The code generation algorithm is a simple code generator uses registers to hold values for the duration of a single basic block. However, all live variables were stored at the end of each block. To save some of these stores and corresponding loads we might arrange to assign registers to frequently used variables and keep these registers consistent across block boundaries (globally). Since programs spend most of their time in inner loops, therefore for the time being, let we know the loop structure of a flow graph, and that we know what values computed in a basic block are used outside that block.

One strategy for global register allocation is to assign some fixed number of registers to hold the most active values in each inner loop. Registers not already allocated may be used to hold values local to one block as in a simple code generator. This approach has the drawback that the fixed number of registers is not always the right number to make available for global register allocation.

User Counts – A simple method for determining the savings to be realized by keeping variable x in a register for the duration of loop L is to recognize that in our machine model we save one unit of cost for each reference to x if x is in a register.

We count a savings of one for each use of x in loop L that is not preceded by an assignment to x in the same block. We also save two units if we can avoid a store of x at the end of a block. Thus if x is allocated a register, count a savings of two for each block of L for which x is live on exit and in which x is assigned a value.

Register Assignment for Outer Loops – Having assigned registers and generated code for inner loops, we may apply the same idea to progressively larger loops. If an outer loop L_1 contains an inner loop L_2 , the names allocated registers in L_2 need not be allocated registers $L_1 - L_2$. However, if the name x is allocated a register in loop L_1 but not L_2 , we must store x on entrance to L_2 and load x if we leave L_2 and enter a block of $L_1 - L_2$.

Register Allocation by Graph Coloring – When a register is needed for a computation but all available registers are in use, the contents of one of the used registers must be stored (spilled) into a memory location in order to free up a register. Graph coloring is a simple, systematic technique for allocating registers and managing register spills.

In this method two passes are used. In the first, target machine instructions are selected as though there were an infinite number of symbolic registers; in effect, names used in the intermediate code become names of registers and three-address statements become machine-language statements. If access to variables requires instructions that use stack pointers, display pointers, base registers or other quantities, then we assume that these quantities are held in registers reserved for each purpose. If access is more complex, the access must be broken into several machine instructions.

Once the instruction have been selected, a second pass assigns physical registers to symbolic ones.

In the second pass, for each procedure a *register-interference graph* is constructed in which the nodes are symbolic registers and an edge connects two node if one is live at a point where the other is defined.

An attempt is made to color the register-interference graph using k colors; where k is the number of assignable registers. A color represents a register, and the coloring makes sure that no two symbolic registers that can interfere with each other are assigned the same physical register. A graph is said to be *colored* if each node has been assigned a color in such a way that no two adjacent nodes have the same color.

NUMERICAL PROBLEMS

Prob.16. Show the annotated parse tree and code generation process for the arithmetic expression $a + (b - c) + d$. (R.G.P.V., Dec. 2012)

Sol. To draw an annotated parse tree, we need the attribute values at each node of the parse tree. Consider the following grammar for the given integer arithmetic expression –

$$\begin{aligned} E &\rightarrow E + T \mid E - T \mid T \\ T &\rightarrow (E) \mid \text{num} \end{aligned}$$

For E and T productions, the semantic rule computes the values of attribute val for the nonterminal on the left side from the values of val for the nonterminals on the right side.

needed outside the block. The interior nodes of the DAG can be evaluated in any order that is a topological sort of the DAG. In a topological sort, a node is not evaluated until all of its children that are interior nodes have been evaluated. As we evaluate a node, its value is assigned to one of its attached identifiers. As preferring one whose value is needed outside the block.

If there are additional attached identifiers y_1, y_2, \dots, y_k for a node n whose values are also needed outside the block, these are assigned with statements $y_1 := x, y_2 := x, \dots, y_k := x$. If n has no attached identifiers at all a new temporary name is created to hold the value of n .

The basic block statements can be reduced to some extent by taking advantage of the common subexpressions exposed during the DAG construction process and by eliminating unnecessary assignments.

Q.41. Explain the various applications of DAG. Construct DAG for the following basic block –

$D := B * C$
 $E := A - ?$
 $B := B * C$
 $A := E - ?$

Ans. Refer to Q.-0 and Prop.18

(A.G.P.), June 2011

Q.42. Explain the heuristic ordering algorithm for DAG

(A.G.P.), Dec. 2005, June 2007, 2011

Ans. In selecting an ordering for the nodes of a DAG we are only constrained to be sure that the order preserves the edge relationships of the DAG. These are the relationships between nodes and their children or parents. The order of evaluation of the nodes is determined by the order of the edges. The order of evaluation of the nodes is determined by the order of the edges. The order of evaluation of the nodes is determined by the order of the edges.

Heuristic Ordering Algorithm

- (i) while unlisted interior nodes remain do begin
- (ii) select an unlisted node n , all of whose parents have been listed;
- (iii) list n ;
- (iv) while the leftmost child m of n has no unlisted parents and is not a leaf do
 - /* since n was just listed, m is not yet listed */
 - begin
 - (v) list m ;
 - (vi) $n := m$
 - end
- end

Fig. 4.29 Heuristic Ordering Algorithm

Q.43. How basic blocks are represented through DAG?

Ans. Directed Acyclic Graphs (DAGs) are useful data structures for implementing transformations on basic blocks. A DAG represents how the value computed by each statement in a basic block is used in subsequent statements of the block. Construction of a DAG from three-address statements is useful for determining common subexpressions within a block, determining which names are used inside the block but evaluated outside the block and determining which statements of the block have their computed value used outside the block. A DAG for a basic block is a Directed Acyclic Graph with the following labels on nodes –

- (i) Leaves are labeled by unique identifiers, either variable names or constants. The leaves represent initial values of names subscripted with 0.
- (ii) Interior nodes are labeled by an operator symbol.
- (iii) Nodes are optionally given a sequence of identifiers for labels. The intention is that interior nodes represent computed values and the identifiers labeling a node are deemed to have that value.

Each node of a flow graph can be represented by a DAG for example,

- (i) $t_1 := 4 * i$
- (ii) $t_2 := a[t_1]$
- (iii) $t_3 := 4 * i$
- (iv) $t_4 := b[t_3]$
- (v) $t_5 := t_2 * t_4$
- (vi) $t_6 := \text{prod} + t_5$
- (vii) $\text{prod} := t_6$
- (viii) $t_7 := i + 1$
- (ix) $i := t_7$
- (x) if ($i \leq 20$) goto (1)

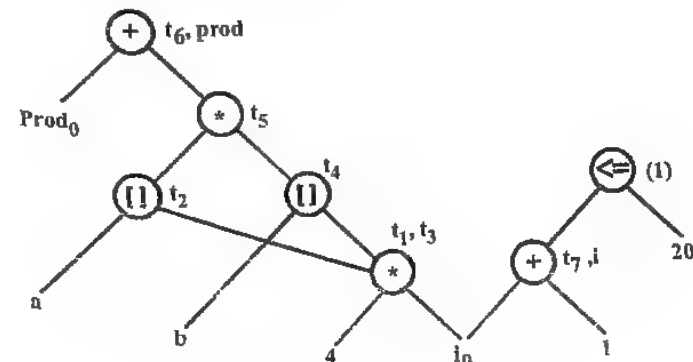


Fig. 4.30 DAG for the Block

Fig. 4.30 represents the DAG of a given basic block. The first statement is $t_1 := 4 * i$, leaves are created and labeled 4 and i. In step (ii), a node is created with labeled *, in step (iii) identifier t_1 is attached to it. For second statement, $t_2 := a[t_1]$, a new leaf is created with label 'a' and find created node t_1 and a new node labeled [] is created and attach with a & t_1 as children. Similarly all the statements are represented by the DAG.

Q.44. Discuss the issues in the design of code generator. Also discuss the DAG representation of basic blocks.
(R.G.P.V., May 2019)

Ans. Refer to Q.29 and Q.43.

Q.45. What is DAG ? What are its advantages in context of optimization ? How does it help in elimination of common sub-expressions ?

Ans. DAG – Refer to Q.38.

Advantages – Refer to Q.40.

Elimination of Common Sub-expressions by Using DAG – The common sub-expressions in a basic block can be automatically detected if we construct a directed acyclic graph (DAG). If the interior nodes in the DAG have more than one label, then those nodes of DAG represent the common sub-expressions in the basic block. After detecting these common sub-expressions, we eliminate them from the basic block. The following example shows the elimination of common sub-expressions and the DAG is shown in fig. 4.31.

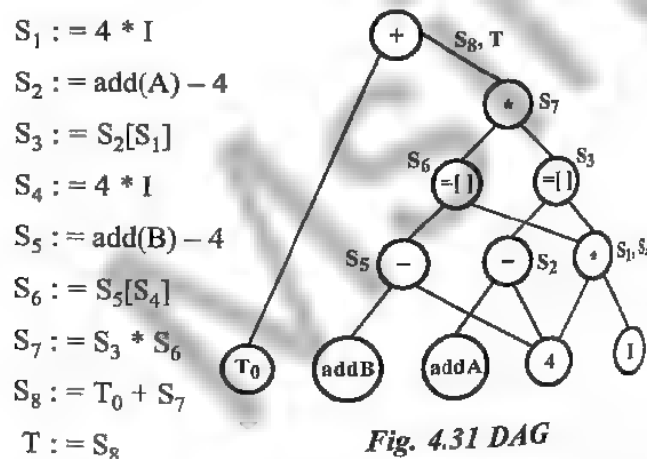


Fig. 4.31 DAG

In fig. 4.31, T_0 is the initial value of T . We see that the value assigned to S_8 is the same as T . Similarly, the values computed for S_1 and S_4 are the same. Therefore, we can eliminate these common sub-expressions by selecting one of the attached identifiers. The reduced basic block can be

written as -

$$S_1 := 4 * I$$
$$S_7 := \text{add}(A) - 4$$
$$S_3 := S_2[S_1]$$
$$S_5 := \text{add}(B) - 4$$
$$S_6 := S_5[S_1]$$
$$S_7 := S_3 * S_6$$
$$T := T_0 + S_7$$

Q.46. What is peephole optimization? Explain it. (R.G.P.V., Nov. 2018)

Or

Write a short note on peephole optimization.

(R.G.P.V., Dec. 2002, 2003, June 2004, Dec. 2004, 2005, 2006, June 2007, Dec. 2008, 2011, 2016, 2017)

Or

Explain briefly peephole optimization.

(R.G.P.V., Dec. 2007, 2010, June 2011, Dec. 2015)

Or

Describe peephole optimization briefly.

(R.G.P.V., Dec. 2013)

Or

What do you mean by peephole optimization ?

(R.G.P.V., June 2005, 2006, 2017)

Of

Give a brief note on peephole optimization. (R.G.P.V., May 2018)

Ans. A target code, generated by a statement-by-statement code-generation strategy, contains redundant instruction and suboptimal constructs. Peephole optimization is the technique used for locally improving the intermediate or object or target code. It is a method for trying to improve the performance of the target program by examining a short sequence of target instruction (called the peephole) and replacing these instructions by a shorter or faster sequence. The peephole is small, moving window on the target program. The code in the peephole need not be contiguous. It is characteristic of peephole optimization that each improvement may spawn opportunities for additional improvements. Thus repeated passes over the code are necessary to get the maximum benefit from peephole optimization. The characteristics of peep-hole optimization are—

(i) **Redundant-instruction Elimination** – Example of redundant loads and stores like the instruction sequence.

- (a) MOV R0,a
(b) MOV a,R0

The instruction (b) can be deleted because whenever (b) is executed, (a) will ensure that a value of a is already in register R0. The instruction (b) is not removed when (b) had a label that means it is not sure that (a) was always executed immediately before (b). For this transformation (a) and (b) have to be in the same basic block.

(ii) **Unreachable Code** – Peephole optimization removes the unreachable instructions. An unlabeled instruction immediately following an unconditional jump may be removed. This can be repeated to eliminate a sequence of instructions. For example, for debugging purposes, a large program may have within it certain segments that are executed only if DEBUG is 1. The intermediate code for if-statement may be –

```
if DEBUG = 1 goto L1
goto L2
L1 : Print debugging aids
L2 :
```

One immediate peephole optimization is to eliminate jumps over jumps. That is, above code is replaced by –

```
if DEBUG ≠ 1 goto L2
Print debugging aids
L2 :
```

It is again optimized as if debug is set to 0.

```
if 0 ≠ 1 goto L2
Print debugging aids
L2 :
```

The argument of the first statement evaluates to a constant true, it can be replaced during constant propagation by goto L₂. Then all the statements that print debugging aids are manifestly unreachable and can be eliminated one at a time.

(iii) **Flow-of-control Optimizations** – The unnecessary jumps can be eliminated in either the intermediate code or the target code by the following types of peephole optimizations –

```
goto L1          goto L2
-----          -----
L1 : goto L2      L1 : goto L2
```

If there are no jumps to L1, then it may be possible to eliminate the statement L1: goto L2 provided it is preceded by an unconditional jump. Then

```
if a < b goto L1    if a < b goto L2
-----            -----
L1 : goto L2        L1 : goto L2
```

Finally, suppose there is only one jump to L1, and L1 is preceded by an unconditional goto. Then,

```
goto L1          if a < b goto L2
-----          -----
L1 : if a < b goto L2  ⇒ goto L3
```

```
L3 :          L3 :
```

(iv) **Algebraic Simplification** – A few algebraic identities occur frequently enough that it is worth considering implementing them.

For example,

$x := x + 0$

or

$x := x * 1$

These are generated by intermediate code generation-algorithms and can be eliminated through peephole optimization.

(v) **Strength Reduction** – Reduction in strength replaces expensive operations by equivalent cheaper ones on the target machine. For example, x^2 is invariably cheaper to implement as $x * x$. Fixed-point multiplication or division by a power of two is cheaper to implement as a shift. Floating point division by a constant can be implemented as multiplication by a constant, which may be cheaper.

(vi) **Use of Machine Idioms** – The target machine may have hardware instructions to implement certain specific operations efficiently. Detecting situations that use these instructions can reduce execution time. For example, some machines have auto-increment and auto-decrement address instructions which improves the quality of code when passing arguments to a function or passing. These modes can be used to pass

Q.47. Describe the process of code generation from DAG

(R.G.P.V., Dec. 2006)

Ans. The advantage of generating code for a basic block from its DAG representation is that it is very easy to rearrange the order of final computation sequence than the linear sequence of three-address statements or quadruples. The following procedure is applicable to generate code from DAG –

(i) **Rearranging the Order** – The order in which computations are done can affect the cost of resulting object code.

For example, we consider the following basic block whose DAG representation is shown in fig. 4.32.

$t_1 := a + b$
 $t_2 := c + d$
 $t_3 := e - t_2$
 $t_4 := t_1 - t_3$

The code generated for the three address statements are –

```

MOV    a,    R0
ADD    b,    R0
MOV    c,    R1
ADD    d,    R1
MOV    R0,   t1
MOV    e,    R0
SUB    R1,   R0
MOV    t1,   R1
SUB    R0,   R1
MOV    R1,   t4
  
```

Fig. 4.33 Code Sequence

If the order of statement can be rearranged then

```

t2 := c + d      MOV c, R0
t3 := e - t2     ADD d, R0
t1 := a + b      MOV e, R1
t4 := t1 - t3    SUB R0, R1
                  MOV a, R0
                  ADD b, R0
                  SUB R1, R0
                  MOV R0, t4
  
```

(a) Basic Block

(b) Revised Code Sequence

Fig. 4.34

(ii) **Heuristic Ordering for DAG's** – The reordering improved the code, i.e. the computation of t_4 was made to follow immediately after the computation of t_1 , its left operand in the tree. The left argument for the computation of t_4 must be in a register for efficient computation of t_4 , and computing t_1 immediately before t_4 . While selecting an ordering for the nodes of a DAG, care should be taken that it preserves the edge relationships of the DAG. The heuristic ordering algorithm which attempts as far as possible to make the evaluation of a node immediately follow the evaluation of its leftmost argument.

(iii) **Optimal Ordering for Trees** – Optimal order means the order that yields the shortest instruction sequence over all instruction sequences that evaluate the tree. The algorithm that determines the optimal order has two parts. The first part labels each node of the tree, bottom-up, with an integer that denotes the fewest number of registers required to evaluate the tree with

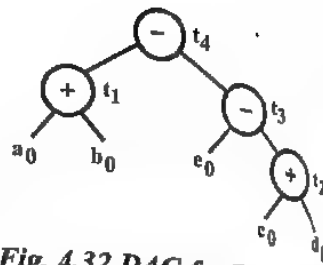


Fig. 4.32 DAG for Basic Block

no stores of intermediate results. The second part is a tree traversal whose order is governed by the computed node labels. The output code is generated during tree traversal.

(iv) **Labeling Algorithm** – The labeling can be done by visiting nodes in a bottom-up order so that a node is not visited until all its children are labeled. In an important special case if n is a binary node and its children have labels l_1 and l_2 , then

$$\text{label}(n) = \begin{cases} \max(l_1, l_2) & \text{if } l_1 \neq l_2 \\ l_1 + 1 & \text{if } l_1 = l_2 \end{cases}$$

Label Computation Algorithm for Postorder Traversal –

- (i) if n is a leaf then
- (ii) if n is the leftmost child of its parent then
- (iii) label(n) := 1
- (iv) else label(n) := 0
- else /* n is an interior node */
- begin
- (v) let n_1, n_2, \dots, n_k be the children of n ordered by label, so label(n_1) \geq label(n_2) $\geq \dots \geq$ label(n_k);
- (vi) label(n) := max(label(n_i) + i - 1)
- $1 < i < k$

end

For example, in the post order traversal of the given tree the nodes are visited in the order $a \ b \ t_1 \ e \ c \ d \ t_2 \ t_3 \ t_4$. Node a is labeled 1 since it is a left leaf and node b is labeled 0 since it is a right leaf. Node t_1 is labeled 1 because the labels of its children are unequal and the maximum label of a child is 1. t_4 and t_3 are labeled with 2 as two registers are needed to evaluate.

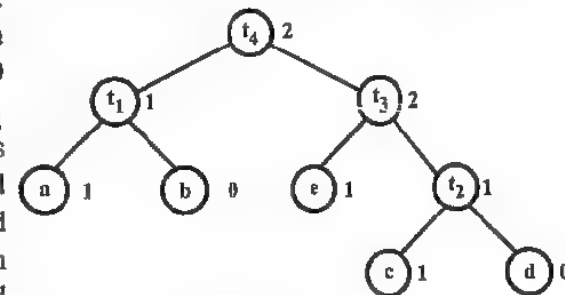


Fig. 4.35 Labeled Tree

(v) **Code Generation from a Labeled Tree** – Generate code for the labeled tree in fig. 4.35 with rstack = R0, R1 initially.

```

gencode(t4)      [R0R1]
gencode(t3)      [R1R0]
gencode(e)       [R1R0]
Print MOV e,R1
  
```

```

gencode (t2)           [R0]
gencode (c)             [R0]
Print MOV c, R0
Print ADD d, R0
Print SUB R0, R1
gencode(t1)           [R0]
gencode(a)              [R0]
Print MOV a, R0
Print ADD b, R0
Print SUB R1, R0

```

(vi) **Multiregister Operations** – The operations like multiplication, division or a function call require more than one register to perform. If a function call require all r registers, replace algorithm line (vi) by label (n) = r . If multiplication requires two registers then

$$\text{label}(n) = \begin{cases} \max(2, l_1, l_2) & \text{if } l_1 \neq l_2 \\ l_1 + 1 & \text{if } l_1 = l_2 \end{cases}$$

where, l_1 and l_2 are the labels of the children of n .

(vii) **Algebraic Properties** – A given tree can be replaced by a simpler one with smaller labels and/or fewer left leaves by making use of algebraic laws. For example, since $+$ is normally regarded as being commutative, we may replace the tree of fig. 4.36 (a) by that of fig. 4.36 (b), reducing the number of left leaves by one and possibly lowering some labels as well. The nodes labeled $+$ in fig. 4.36 (c) and replace it by a left chain as in fig. 4.36 (d).

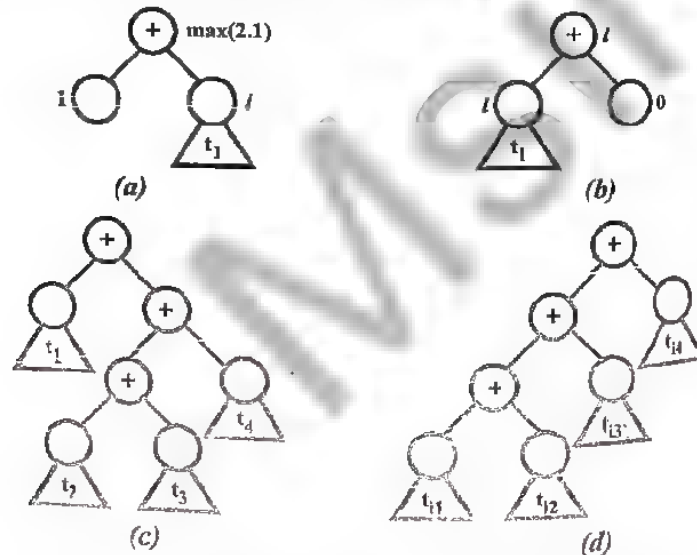


Fig. 4.36 Commutative and Associative Transformation

(viii) **Common Subexpressions** – The common subexpressions will correspond to nodes with more than one parent called shared nodes. In fact, common subexpressions make code generation more difficult.

A reasonable solution is to partition the DAG into a set of trees by finding for each root and/or shared node n the maximal subtree with n as root that includes no other shared nodes, except as leaves. After partitioning DAG into trees, the ordering of the evaluation of trees is done and any of the preceding algorithms is used to generate code for each tree.

Q.48. What are the advantages of DAG? Describe the process of code generation from DAG (R.G.P.V., Dec. 2010)

Ans. Refer to Q.40 and Q.47.

NUMERICAL PROBLEMS

Prob.17. Draw DAG for the given block.

```

a = b + c
b = a - d
c = b + c
d = a - d

```

(R.G.P.V., Dec. 2015)

Sol.

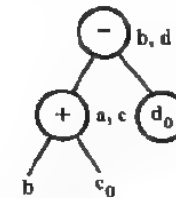


Fig. 4.37 DAG for Basic Block

Prob.18. Construct a DAG for the basic block whose code is given below –

```

D := B * C
E := A + B
B := B * C
A := E - D

```

(R.G.P.V., Dec. 2003, June 2004, Dec. 2005, June 2007, Dec. 2016, Nov. 2018)

Or

Construct the DAG for –

```

X = Y * Z
W = P + Y
Y = Y * Z
P = W - X

```

(R.G.P.V., Dec. 2012)

Sol.

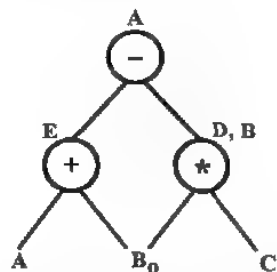


Fig. 4.38 DAG for Basic Block

Prob.19. Construct the DAG for the following basic block –

$$d := b * c$$

$$e := a + b$$

$$b := b * c$$

$$a := e - d$$

Then generate the code for the above constructed DAG using only one register.
(R.G.P.V., Dec. 2011)

Sol. DAG – Refer to Prob.18.

Code for the DAG using only one register is –

$$\text{MOV } b, R_0$$

$$\text{MUL } c, R_0$$

$$\text{MOV } R_0, d$$

$$\text{MOV } a, R_0$$

$$\text{ADD } b, R_0$$

$$\text{SUB } d, R_0$$

$$\text{MOV } R_0, a$$

Prob.20. Construct DAG for the following basic block

$$d := b + c$$

$$e := a + b$$

$$b := b * c$$

$$a := e - d$$

Sol.

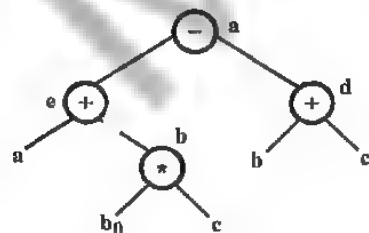


Fig. 4.39 DAG for Basic Block

(R.G.P.V., June 2017)

Prob.21. Construct DAG for given expression $(x + 5) * (x + 5 + y)$.

(R.G.P.V., June 2016)

Sol. The three address code for the given expression is –

$$t_1 = x + 5$$

$$t_2 = t_1 + y$$

$$t_3 = t_1 * t_2$$

DAG representation is shown in fig. 4.40.

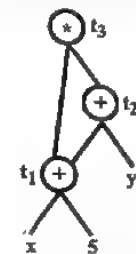


Fig. 4.40

Prob.22. Construct DAG for the following expression

$$a + a * (b - c) + (b - c) * d$$

(R.G.P.V., Dec. 2014)

Sol. The three address code for the given expression is

$$t_1 = b - c$$

$$t_2 = a * t_1$$

$$t_3 = t_1 * d$$

$$t_4 = a + t_2$$

$$t_5 = t_4 + t_3$$

DAG representation is shown in fig. 4.41.

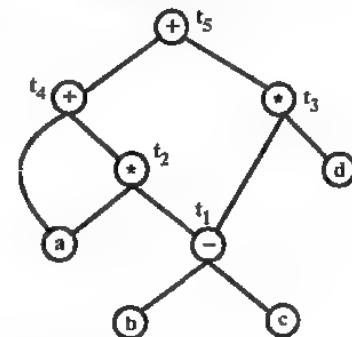


Fig. 4.41

Prob.23. Show the DAG for the following statement –

$$Z = X - Y + X * Y * U - V/W + X + V$$

(R.G.P.V., June 2008, 2012)

Sol. The three address code for the given expression is

$$t_1 = X * Y$$

$$t_2 = t_1 * U$$

$$t_3 = X - Y$$

$$t_4 = t_3 + t_2$$

$$t_5 = V/W$$

$$t_6 = t_4 - t_5$$

$$t_7 = t_6 + X$$

$$t_8 = t_7 + V$$

$$Z = t_8$$

DAG representation is shown in fig. 4.42.

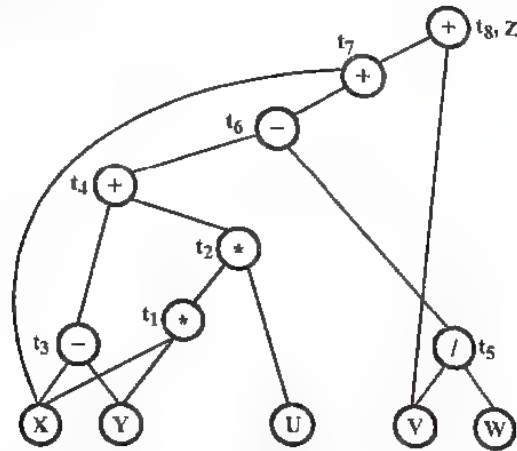


Fig. 4.42

Prob.24. Draw the syntax tree and DAG for the expression –
 $(a * b) + (c - d) * (a * b) + b$

(R.G.P.V., Dec. 2011)

Sol. The syntax tree for the given expression is shown in fig. 4.43.

Three address code for the given expression is

$t_1 = a * b$
 $t_2 = c - d$
 $t_3 = t_2 * t_1$
 $t_4 = a * b$
 $t_5 = t_4 + t_3$
 $t_6 = t_5 + b$

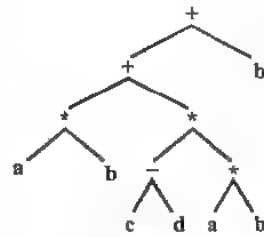


Fig. 4.43 Syntax Tree

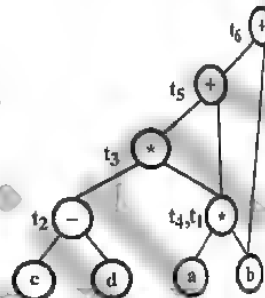


Fig. 4.44 DAG

DAG representation is shown in fig. 4.44.

Prob.25. Construct the DAG for the following basic block –

$a := b + c$
 $b := b - d$
 $c := c + d$
 $e := b + c$

Sol.

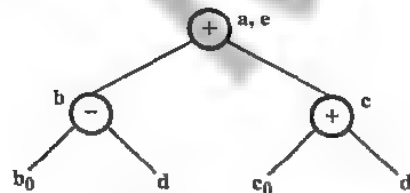


Fig. 4.45 DAG for Basic Block

(R.G.P.V., May 2018)

UNIT

5

CODE OPTIMIZATION

INTRODUCTION TO CODE OPTIMIZATION – SOURCES OF OPTIMIZATION OF BASIC BLOCKS, LOOPS IN FLOW GRAPHS, DEAD CODE ELIMINATION, LOOP OPTIMIZATION

Q.1. Write a short note on code optimization.

Or

What is code optimization? How is it achieved? Explain.

(R.G.P.V., Dec. 2013)

Ans. The term “code optimization” refers to the technique, a compiler can employ in an attempt to produce a better object language program than the most obvious for a given source program. Code optimization aims for improving the execution efficiency of a program. This is achieved in two ways –

- Redundancies in a program are eliminated.
- Computations in a program are rearranged or rewritten to make it efficient code to gain the advantage of execution speed without changing the meaning of a program.

The quality of an object program is generally measured by its size or the running time. For large computations, running time is particularly important. For small computations, size may be as important as time. It is theoretically impossible for compiler to provide best possible object program for every source program under any reasonable cost function. Therefore, a more accurate term for code optimization would be “code improvement”. Code optimization must not change the meaning of a program. Optimization seeks to improve a program than the algorithm used in a program. Thus replacement of an algorithm by a more efficient algorithm is beyond the scope of optimization, efficient code generation for a specific target machine is also beyond its scope. The optimization techniques are independent of the target machine.

Compilers that apply code improving transformations are called optimizing compilers. Optimizing compilers make only well judged attempts to improve the code it produces. It improve the code without costing too much time at

compilation. The optimizer can also be defined as a system which transforms an program input to it into a program which is semantically equivalent to the original program, and which executes more efficiently. Optimizing compiler, in which the input program is written in an HLL, while the output program could be in an HLL or in low level language. It is not portable.

A source-to-source optimizer has the advantages that its output is in an HLL, and thus it is portable. However, source-to-source optimizers are not very widely used because optimization of array references and optimal utilization of machine register, which together account for the major gains of optimization, are not possible at the source code level.

For rearranging the computations in a program with an eye on execution efficiency, the optimizer has to exploit –

- (i) Characteristics of the target machine in terms of the addressing structure and instruction repertoire.
- (ii) Characteristics of a program in terms of program structure and nature of the computations involved.

Q.2. Explain why code optimization is called optional phase.

(R.G.P.V., June 2010)

Ans. Code optimization is an optional phase used to improve the intermediate code so that the ultimate object program runs faster and/or takes less space. Its output is another intermediate code program that does the same job as the original, but perhaps in a way that saves time and/or space.

Q.3. What are the properties for code-improving transformation ?

Ans. The best transformations are those that yield the most benefit for the least effort. The transformations provided by an optimizing compiler should have several properties –

(i) A transformation must preserve the meaning of programs. That is, an “optimization” must not change the output produced by a program for a given input, or cause an error, such as a division by zero, that was not present in the original version of the source program. The influence of this criterion prevades code optimization at all times we take the “safe” approach of missing an opportunity to apply a transformation rather than risk changing what the program does.

(ii) A transformation must, on the average, speed up programs by a measurable amount. Sometimes we try to reduce the space taken by the compiled code, although the size of code has less importance than it once had. Even though every transformation do not succeeds in improving every program, and occasionally on “optimization” may slow down a program slightly, as long as on the average it improves things.

(iii) A transformation must be worth the effort. It does not make a sense for a compiler writer to expend the intellectual effort to implement a code-improving transformation and to have the compiler expend the additional time compiling source programs if this effort is not repaid when the target programs are executed.

Some transformations can only be applied after detailed, often time consuming, analysis of the source program, so there is a little point in applying them to programs that will be run only a few times. For example, a fast, non-optimizing, compiler is likely to be more helpful during debugging or for “student jobs” that will be run successfully a few times and thrown away.

Q.4. Explain the principle sources of optimization with suitable example.
(R.G.P.V., June 2010, Dec. 2010)

Or

Write short note on principle sources of optimization.

(R.G.P.V., June 2016)

Ans. An optimizing transformation is a rule for rewriting a segment of a program to improve its efficiency without affecting its meaning. Optimizing transformations are classified into local and global depending on whether they are applied over small segment of a program consisting of a few source statements or over larger segments consisting of loops or function bodies. A few optimizing transformation used in compiler are –

(i) **Function Preserving Transformation** – Function preserving transformation, in which a compiler can improve a program, without changing the function it computes. Some of the function preserving transformation are –

(a) **Common Subexpression Elimination** – Common subexpression elimination implies the elimination of an expression’s evaluation from a place in the program, if an equivalent value has already been computed and can be used, common subexpression or equivalent expressions are occurrences of expressions yielding the same value. For example,

$A = B * * C$	$T = B * * C$
$J = J - 1$	$A = T$
$M = 4 * i + B * * C$	$J = J - 1$
$Y = a[M]$	$M = 4 * i + T$
	$Y = a[M]$
$X = B * * C + 5.2$	$X = T + 5.2$

Two expressions are said to be ‘common’ if –

- (1) The expressions are congruent, i.e., they consist of identical operands connected to each other by identical operators.
- (2) The expressions are equivalent, i.e, no potential definitions for any of their operands exist between the evaluations of these expressions.

(b) **Copy Propagation** – The assignment of the form $f := g$ called copy statements. In this transformation g is used for f . For example,

$x := t_3$	$x := t_3$
$a[t_2] := t_5$	$a[t_2] := t_5$
$a[t_4] := x$	$a[t_4] := t_3$
$t_6 := a[t_9]$	$t_6 := a[t_9]$
$a[t_9] := x$	$a[t_9] := t_3$

(c) **Dead Code Elimination** – Code which can be omitted from a program without affecting its results is called **dead code**. A piece of code is said to be 'Dead' if the results of evaluating the code are not used anywhere in the program. Such code can be eliminated safely. Dead code is detected by checking whether the value assigned in an assignment statement is used anywhere in the program.

```
ITEMP = 5;
I = 1;
LOOP: If ITEMP > 105 THEN GOTO NEXT;
      D = ITEMP;
      ITEMP = ITEMP * 10;
      I = I + 2;
      GOTO LOOP;
NEXT : .....
```

In above code 'I' would become redundant if it were not used anywhere else in the loop body. In above case, the initialization statement $I = 1$; and the incrementation $I = I + 2$; can both be eliminated.

(d) **Constant Folding or Compile Time Evaluation** – Execution efficiency can be improved by performing certain actions during compilation itself. This reduces the execution time of the program. Constant folding of operation means when all operands are constants the operation can be evaluated at compilation time itself. The result of operation is also a constant can replace the original evaluation in the program. For example,

$a := 3.14157/2$ $a := 1.570785$

Constant propagation enhances the possibility of applying folding optimization within a program. If a variable is assigned the value of a constant and the variable is used in an expression before undergoing any other assignment to it, then constant can be used in place of the variable in the expression. For example,

$A = 2.5$	$A = 2.5$
-----	-----
-----	-----
$X = A * 3.1$	$X = 7.75$

(ii) **Loop Optimization** – Loops are important place for optimization. The running time of a program may be improved if we decrease the number of instructions in an inner loop, even the code outside the loop is increased. Three techniques are important for loop optimization –

(a) **Code Motion** – An important modification that decreases the amount of code in a loop, is code motion. Code movement optimization seeks to improve the execution time characteristics of a program by moving the evaluation of an expression to other parts of the program. This takes an expression that yields the same result independent of the number of times a loop is executed (loop invariant computation) and places the expression before the loop. For example,

while ($i \leq \text{limit} - 2$)

becomes,
 $t = \text{limit} - 2$;
 while($i \leq t$)

Frequency reduction transformation is also a code motion technique in which the execution time can be reduced by moving code from a part of a program which is executed very frequently to another part of the program which is executed fewer times, for example,

FOR $i := 1$ TO 100 DO	$X := 25 * a + b$;
BEGIN	FOR $i := 1$ TO 100 DO
$Z := i$;	BEGIN
$X := 25 * a + b$;	$Z := i$;
$Y := X + Z$;	$Y := X + Z$;
END;	END;

\Rightarrow

(b) **Induction Variables** – The values of J and t_4 in the following example remain in lock step, every time the value of J decreases by 1, that of t_4 decreases by 4 because $4 * J$ is assigned to t_4 . Such identifiers are called induction variables. When there are two or more induction variables in a loop it may be possible to get rid of all but one, by the process of induction variable elimination. For example,

$J := J - 1$	
$t_4 := 4 * J$	$t_4 := t_4 - 4$
$t_5 := a[t_4]$	$t_5 := a[t_4]$

(c) **Strength Reduction** – The strength reduction optimization replaces the occurrence of a time consuming operation (a 'high strength' operation) by an occurrence of a faster operation (a 'low strength' operation) e.g. replacement of a multiplication by an addition. Strength reduction is very important for array accesses occurring within program loops. For example,

```

FOR i := 1 TO 10 DO      itemp := 5
BEGIN                    FOR i := 1 TO 10 DO
  Z := i;                begin
  K := i * 5;             Z := i;
  Y := X + Z;             K := itemp;
  -----               Y := X + Z;
  END;                   itemp := itemp + 5;
                        END;

```

Q.5. Write short note on function preserving transformations.

(R.G.P.V., Dec. 2013)

Ans. Refer to Q.4 (i).

Q.6. What are dead code elimination and strength reduction transformation? Illustrate with an example.

(R.G.P.V., May 2019)

Ans. Refer to Q.4 (i) (c) and 4 (ii) (c).

Q.7. Explain dead code elimination with example. (R.G.P.V., Dec. 2012)

Or

Write short note on dead code elimination. (R.G.P.V., Dec. 2016, Nov. 2018)

Or

Analyse the possible causes of a dead code. Explain with an example how the compiler can catch the presence of a dead code. (R.G.P.V., Dec. 2017)

Ans. Refer to Q.4 (i) (c).

Q.8. Explain the following optimizing transformations with suitable examples –

(i) Code movement (ii) Common subexpression elimination.

Or (R.G.P.V., Dec. 2007, June 2011)

Write short notes on the following –

(i) Code movement (ii) Common subexpression elimination.

(R.G.P.V., Dec. 2011)

Ans. (i) Code Movement – Refer to Q.4 (ii) (a).

(ii) Common Subexpression Elimination – Refer to Q.4 (i) (a).

Q.9. Describe the terms common subexpression elimination and dead code elimination in the context of code optimization. (R.G.P.V., June 2016)

Ans. Refer to Q.4 (i) (a) and (c).

Q.10. Explain loop optimization with example. (R.G.P.V., Dec. 2014)

Or

What is loop optimization?

Or

Write short note on loop optimization. (R.G.P.V., Dec. 2016, Nov. 2018)

Ans. Refer to Q.4 (ii).

Q.11. Explain the need of code optimization. With example, illustrate loop optimization. (R.G.P.V., June 2012)

Ans. Need of Code Optimization – Refer to Q.1.

Loop Optimization – Refer to Q.4 (ii).

Q.12. Explain the following with example –

(i) Strength reduction

(ii) Variable propagation

(iii) Common subexpression elimination.

(R.G.P.V., Dec. 2014)

Ans. (i) Strength Reduction – Refer to Q.4 (ii) (c).

(ii) Variable Propagation – In variable propagation, a variable is replaced by another variable having identical value. For example, consider the following statements –

```

X = Y
A = X*B
C = Y*B

```

The statement $X = Y$ specifies that the values of X and Y are equal. Since the value of X or Y is not modified further, the second statement can hence be written as $A = Y + B$ by propagating the variable Y to it. This propagation makes $Y*B$ as a common subexpression in the last two statements, and hence possibly evaluated only once.

(iii) Common Subexpression Elimination – Refer to Q.4 (i) (a).

Q.13. Describe the necessary and sufficient condition for performing constant propagation and dead code eliminations.

(R.G.P.V., June 2009, 2011, Dec. 2011)

Ans. Constant Propagation – Constant propagation is a technique used to enhance the possibilities of applying folding optimization within a program. According to this technique, if a variable is assigned the value of a constant, and the variable is used in an expression before undergoing any other assignment to it, then the constant can be used in place of the variable in the concerned expression. This can lead to folding optimization as in the following example –

```

A = 2.5
-----
-----
X = A*3.1

```

wherein $A*3.1$ can be replaced by 7.75. In practice, constant propagation can also lead to elimination of certain redundant code, as in the following example –

```

I = 3
J = I/2*K
IF (LEQ.3) GOTO 10

```

⇒

```

I = 3
J = K
GOTO 10

```

For applying constant propagation, it is necessary to ascertain that along all paths leading to an expression in which the variable is used, the same constant value is assigned to the variable

Dead Code Elimination – Refer to Q.4 (i) (c).

Dead code elimination is not restricted in scope to situations arising from strength reduction and loop-test replacement alone. An assignment to a variable results in dead code if another assignment is always made to that variable before it is used anywhere in the program.

Q.14. What do you mean by local optimization and global optimization?
Or
Compare local optimization with global optimization.

(R.G.P.V., June 2016)
(R.G.P.V., June 2017)

Ans. It is possible to perform optimizing transformations like common subexpression elimination, copy propagation, dead code elimination, etc. that are localized to a basic block. These optimizing transformation can be arrived at by analyzing the intermediate code of the basic block in isolation. This type of optimization wherein both the analysis and transformations are localized to a basic block is called local optimization. The transformations in local optimization are known as **local transformations**. The name of transformation is usually prefixed with local while referring to the local transformation. For example, local common subexpression elimination, local copy propagation and so on. If we construct a directed acyclic graph (DAG), the local common subexpressions in a basic block can be automatically detected. Local optimization improved the execution efficiency of programs by a factor of 1.4 on the average.

The scope of local optimization is limited to small parts of program code. Therefore, only short code sequences are considered together to explore the possibility of optimization. This restricts the amount of program analysis needed for the purpose of optimization. However, it also necessarily restricts the gains of optimization.

The optimization wherein both the analysis and the transformations of the three address code spanning across multiple basic blocks of a procedure is called global optimization. The transformations in global optimization are known as global transformations. The name of transformation is usually prefixed with global while referring to the global transformation. For example, global common subexpression elimination, global copy propagation and so on. In the optimization phase of a compiler, the global transformations usually follow the local transformations. The locally optimized code is taken as input for global optimization. To detect global common subexpression, we need to compute available expressions. Global optimization improves the execution efficiency by a factor of 2.7 or above.

The scope of global optimization is not limited to short code sequences. Therefore, optimization of large part of program can be done, including the loops. This requires considerable analysis of the program structure and its gains are also higher than those of local optimization.

Q.15. Write short note on local and loop optimization.

(R.G.P.V., May 2018)

Ans. Refer to Q.14 and Q.4 (ii).

Q.16. Compare the types of optimization that should be applied before code generation.

(R.G.P.V., Dec. 2009)

Ans. Optimization can be machine independent or machine dependent. Machine-independent optimizations can be performed independently of the target machine for which the compiler is generating code. That is, the optimizations are not tied to the target machine's specific platform or language. Examples of machine-independent optimizations are—elimination of loop invariant computation, induction variable elimination and elimination of common subexpressions.

On the other hand, machine-dependent optimization requires knowledge of the target machine. An attempt to generate object code that will utilize the target machine's registers more efficiently is an example of machine-dependent code optimization.

Q.17. What are the common algebraic transformations that can be done for improving the intermediate code?

(R.G.P.V., Dec. 2012)

Ans. The useful algebraic transformations are those that simplify expressions or replace expensive operations by cheaper ones. An algebraic identity is a relation that holds true for all values of the symbols involved in it. The common algebraic transformations that can be done for improving the intermediate code are shown in table 5.1.

Table 5.1

Intermediate Code Statement	Name of the Identity	Identity Applied on Statement
$Y = X + 0$	Additive identity	$Y = X$
$Y = X * 1$	Multiplicative identity	$Y = X$
$Y = X * 0$	Multiplication with 0	$Y = 0$

Typically, the algebraic identity is applied on a single intermediate code statement and transformed to a copy statement. In algebraic transformations, intermediate code statements with operators like SUB and MUL are transformed into copy statements. The copy statements lend well for copy propagation and subsequent dead code elimination transformations, which result in fewer intermediate code statements. The reduction in the intermediate code statements leads to improvement in execution speed and lower consumption of memory.

Q.18. Explain different types of optimization.

(R.G.P.V., Dec. 2012)

Ans. Refer to Q.4 and Q.17.

Q.19. What are the properties of optimizing the compilers?

(R.G.P.V., June 2017)

Ans. There are various properties of optimizing the compilers –

- (i) It can minimize or maximize attributes of executable computer program.
- (ii) It takes less time to execute a program.
- (iii) It takes less memory.
- (iv) Code at least as good as an assembler programmer.
- (v) State, robust performance.
- (vi) Architectural strengths fully exploited.
- (vii) Architectural weaknesses fully hidden.
- (viii) Broad efficient support for language features.
- (ix) Instantaneous compilation.

Q.20. Write short note on optimization of basic block.

(R.G.P.V., May 2018)

Ans. Many of the structure-preserving transformations can be implemented by constructing a DAG for a basic block. In other words, there is a node in the DAG for each of the initial values of the variables appearing in the basic block, and there is a node n associated with each statement s within the block. The children of n are those nodes corresponding to statements that are the last definitions prior to s of the operands used by s . None n is labeled by the operator applied at s , and also attached to n is the list of variables for which it is the last definition within the block.

Common subexpressions can be detected by noticing, as a new node m is about to be added, whether there is an existing node n with the same children, in the same order, and with the same operator.

Example – A DAG for the block given below –

$a := b + c$
 $b := a - d$
 $c := b + c$
 $d := a - d$

is shown in fig. 5.1. When we construct the node for the third statement $c := b + c$. We know that the use of b in $b + c$ refers to the node of fig. 5.1 labeled b , because that is the most recent definition of b . Thus, we do not confuse the values computed at statements one and three.

However, the node corresponding to the fourth statement $d := a - d$ has the operator $-$ and the nodes labeled a and d_0 as children. Since the operator and the children are the same as those for the node corresponding to statement two, we do not create this node, but add d to the list of definitions for the node labeled.

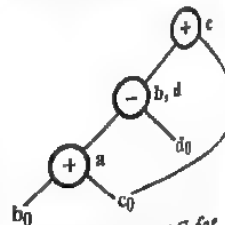


Fig. 5.1 DAG for Basic Block

Q.21. Write a short note on dominators.

Code Optimization 241

Or

Define dominators.

(R.G.P.V., Dec. 2009)

Ans. Let node d of a flow graph *dominates* node n , written $d \text{ dom } n$, if every path from the initial node of the flow graph to n goes through d . Under this definition, every node dominates itself, and the entry of a loop dominates all nodes in the loop.

A useful way of presenting dominator information is in a tree, called the *dominator tree*, in which the initial node is the root, and each node d dominates only its descendents in the tree. For example, fig. 5.3 shows the dominator tree for the flow graph of fig. 5.2.

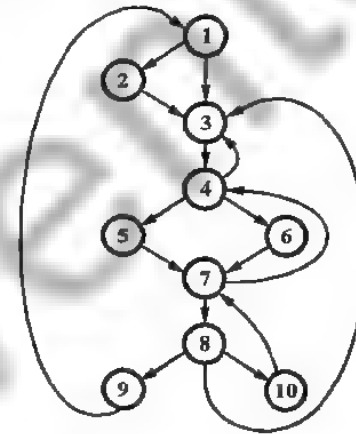


Fig. 5.2 Flow Graph

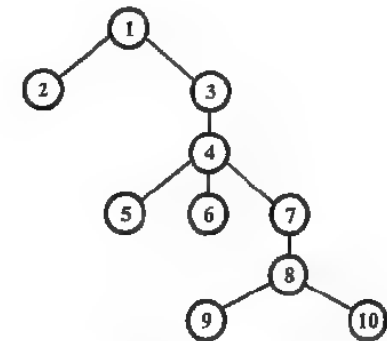


Fig. 5.3 Dominator Tree for Flow Graph of Fig. 5.2

The existence of dominator trees follows from a property of dominators, each node n has a unique *immediate dominator* m that is the last dominator of n on any path from the initial node to n . In terms of the *dom* relation, the immediate dominator m has that property that if $d \neq n$ and $d \text{ dom } n$, then $d \text{ dom } m$.

Q.22. What do you understand by natural loops and inner loops?

Or

Write short note on loops in flow graphs.

(R.G.P.V., Dec. 2013)

Ans. One important application of dominator information is in determining the loops of a flow graph suitable for improvement. There are two essential properties of such loops –

(i) A loop must have a single entry point, called the “header”. This entry point dominates all nodes in the loop, or it would not be the sole entry to the loop.

(ii) There must be atleast one way to iterate the loop, i.e., at least one path back to the header.

A good way to find all the loops in a flow graph is to search for edges in the flow graph whose heads dominate their tails.

If we use the natural loops as “the loops” then we have the useful property that unless two loops have the same header, they are either disjoint or one is entirely contained the other. Thus, neglecting loops with the same header for the moment, we have a natural notion of *inner loop* – one that contains no other loops.

When two loops have the same header, as in fig. 5.4, it is hard to tell which is the inner loop. For example, if the test at the end of B_1 were

if $a = 10$ goto B_2

probably the loop $\{B_0, B_1, B_3\}$ would be the inner loop. However, we could not be sure without a detailed examination of the code. Perhaps a is almost always 10, and it is typical to go around the $\{B_0, B_1, B_2\}$ loop many times before branching to B_3 . Thus, we shall assume that when two natural loops have the same header, but neither is nested within the other, they are combined and treated as a single loop.

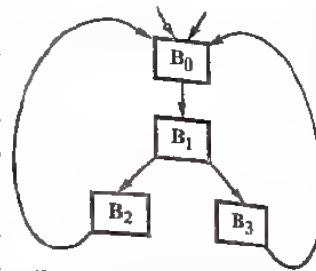


Fig. 5.4 Two Loops with the Same Header

Q.23. Define the following –

- (i) *Dominators* (ii) *Natural loops* (iii) *Inner loop* (iv) *Preheader*.
(R.G.P.V., Dec. 2014)

Ans. (i) Dominators – Refer to Q.21.

(ii) Natural Loops – Refer to Q.22.

(iii) Inner Loop – Refer to Q.22.

(iv) Preheader – A preheader block serves as a placeholder for the quads that need to be executed just before entering the loop. The preheader is a basic block introduced during the loop optimization to hold the quads that are moved from within the loop. It is a predecessor to the header block.

Q.24. Write short note on loop unrolling, peeling, fusion.
(R.G.P.V., Dec. 2013)

Ans. Loop unrolling involves replicating the body of the loop to reduce the required number of tests if the number of iterations are constant. For example consider the following loop –

```

I = 1
while (I <= 100)
{
    x[I] = 0;
    I ++;
}
  
```

Here, the test $I \leq 100$ will be performed 100 times. But if the body of the loop is replicated, then the number of times this test will need to be performed will be 50. After replication of the body, the loop will be –

```

I = 1
while (I <= 100)
{
    x[I] = 0;
    I ++;
    X[I] = 0;
    I ++;
}
  
```

It is possible to choose any divisor for the number of times the loop is executed, and the body will be replicated that many times. Unrolling saves 50% of the maximum possible executions.

When several adjacent loops perform calculations over the same range of values, and the later loops do not depend on a value calculated by the earlier loops, it can often make sense to combine the loops into a single loop; this is called loop fusion.

In some cases, it may not be possible to combine the initial or final iterations of the two loops. In these cases, those iterations are peeled off into separate code. Loop peeling is also useful in situations where the behavior of a loop changes depending on the index variable – perhaps the first iteration of the loop has code that initializes values; this first iteration can be peeled off, reducing the complexity of the rest of the code.

INTRODUCTION TO GLOBAL DATA FLOW ANALYSIS, CODE IMPROVING TRANSFORMATIONS, DATA FLOW ANALYSIS OF STRUCTURE FLOW GRAPH, SYMBOLIC DEBUGGING OF OPTIMIZED CODE

Q.25. Write short note on global data flow analysis.
(R.G.P.V., Dec. 2002, June 2005, 2006, Dec. 2017)

Or

What is global data flow analysis? What is its use in code optimization?
(R.G.P.V., Dec. 2006, June 2008, Dec. 2010, June 2011, Dec. 2011, June 2012, Dec. 2012, 2014, 2015)

Or

Write and explain data flow equations. (R.G.P.V., June 2009, Dec. 2009)

Or

Explain briefly data flow equations.

(R.G.P.V., June 2011)

Or
Write short note on data flow equations.

(R.G.P.V., Dec. 2011)

Or
Explain data flow analysis of structure flow graph. (R.G.P.V., Dec. 2013)

Ans. For code optimization and a good job of code generation, a compiler needs to collect information about the program as a whole and to distribute this information to each block in the flow graph. Data flow information is collected by an optimizing compiler by a process known as **Data Flow Analysis**. Data flow information such as knowing where a variable was last defined before reaching a given block, in order to perform transformations like constant folding and dead code elimination. Data flow information can be collected by setting up and solving systems of equations that relate information at various points in the program. A typical equation has the form

$$\text{out}[s] = \text{gen}[s] \cup (\text{in}[s] - \text{kill}[s])$$

and can be read as “the information at the end of a statement is either generated within the statement, or enters at the beginning and is not killed as control flows through the statement”. These equations are called **Data Flow Equations**. The details of how data flow equations are set up and solved depend on three factors. These factors are –

(i) The notions of generating and killing depend on the desired information, i.e., on the data flow analysis problem to be solved. For some problems, instead of proceeding along with the flow of control and defining $\text{out}[s]$ in terms of $\text{in}[s]$, we proceed backwards and define $\text{in}[s]$ in terms of $\text{out}[s]$.

(ii) Data flows along control paths, data flow analysis is affected by the control constructs in a program.

(iii) There are subtleties that go along with such statements as procedure calls, assignments through pointer variables, and even assignments to array variables.

Point in a program means the position before or after any intermediate language statement. The control reaches the point just before a statement when that statement is about to be executed, and the point after when that statement has just been executed. For example, Block B_1 has four points in fig. 5.5, one before any of the assignments and one after each

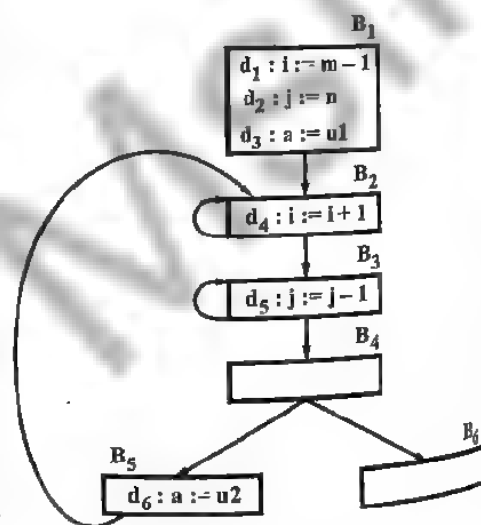


Fig. 5.5 A Flow Graph

of the three assignments. A path from P_1 to P_n is a sequence of points P_1, P_2, \dots, P_n such that for each i between 1 and $n-1$, either

- (i) P_i is the point immediately preceding a statement and P_{i+1} is the point immediately following that statement in the same block, or
- (ii) P_i is the end of some block and P_{i+1} is the beginning of a successor block.

A definition of a variable x is a statement that assigns, or may assign, a value to x . The most common forms of definition are assignments to x and statements that read a value from an I/O device and store it in x . These statements define a value for x , and they are referred to as **unambiguous definitions** of x . The statements that may define a value for x are called **ambiguous definitions**. The forms of ambiguous definitions of x are –

(i) A procedure call with x as a parameter or a procedure that can access x because x is in its scope or x has been identified with another variable that is passed as a parameter or is in the scope.

(ii) An assignment through a pointer that could refer to x .

A definition d reaches a point p if there is a path from the point immediately following d to p , such that d is not “killed” along that path. If a definition d of variable a reaches point p , then d might be the place at which the value of a used at p might last have been defined. A definition of a variable a is killed if between two points along the path there is a definition of a . For example, in the above given fig. 5.5 both the definitions $i := m-1$ and $j := n$ in block B_1 reach the beginning of B_2 . The assignment to j in B_3 kills the definition of $j := n$, so not reach B_4, B_5 or B_6 .

Data Flow Analysis of Structured Programs – Data flow analysis of structured programs is very easy. Flow graphs for control flow constructs such as do-while statements have a property that there is a single beginning point at which control enters and a single end point that control leaves from when execution of the statements is over. For example,

$S \rightarrow id := E / S ; S / \text{if } E \text{ then } S \text{ else } S / \text{do } S \text{ while } E$

$E \rightarrow id + id / id$

The above statement exploit this property and can be represented as

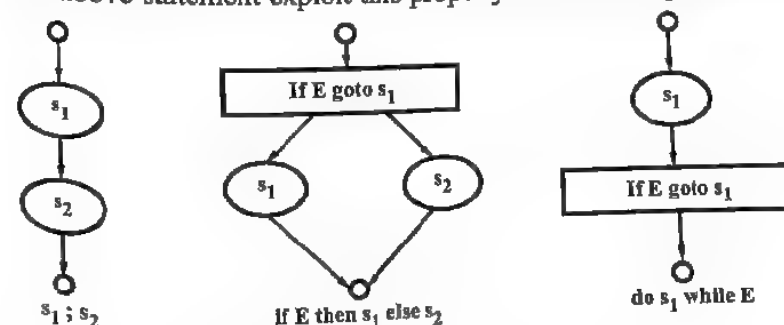


Fig. 5.6 Structured Control Constructs

A portion of a flow graph called a region is defined as a set of nodes N that include a header, which dominates all other nodes in the region. All edges between nodes in N are in the region except that enter the header (A loop is a special case of a region that is strongly connected and includes all its back edges into the header). The dummy blocks with no statements through which control flows just before H enters and just before it leaves the region. The beginning points of the dummy blocks at the entry and exit of a statement's region are the beginning and end points, respectively, of the statement.

Q.26. Write short note on code improving transformation.

(R.G.P.V., Dec. 2013)

Ans. Refer to Q.17.

Q.27. Explain any two data flow properties used by target code generator for producing efficient code.

(R.G.P.V., Dec. 2015)

Ans. Available expression is a data flow property that is computed by the global optimiser using data flow analysis for eliminating the re-evaluation of common subexpressions globally across blocks. Another data flow property is liveness. The liveness information helps us in performing global dead code elimination.

Q.28. Design algorithm for global common subexpression elimination.

(R.G.P.V., Dec. 2016)

Ans. The available expressions data-flow problem allows us to determine if an expression at point p in a flowgraph is a common subexpression. The following algorithm eliminates common subexpressions –

Algorithm – Global common subexpression elimination.

Input – A flow graph with available expression information.

Output – A revised flow graph.

Method – For every statement s of the form $x := y + z$ such that $y + z$ is available at the beginning of s 's block, and neither y nor z is defined prior to statement s in that block, do the following –

(i) To discover the evaluations of $y + z$ that reach s 's block, we follow flow graph edges, searching backward from s 's block. However we do not go through any block that evaluates $y + z$. The last evaluation of $y + z$ in each block encountered is an evaluation of $y + z$ that reaches s .

(ii) Create a new value u .

(iii) Replace each statement $w := y + z$ found in (i) by

$u := y + z$

$w := u$

(iv) Replace statement s by $x := u$.

Some comments on this algorithm are as follows –

(i) The search in step (i) of the algorithm for the evaluation of $y + z$ that reach statement s can also be formulated as a data-flow analysis problem. However, it does not make sense to solve it for all expressions $y + z$ and all statements or blocks, because too much irrelevant information is gathered.

(ii) Not all changes made by this algorithm are improvements. We might wish to limit the number of different evaluations reaching s found in step (i), probably to one. However, copy propagation often allows benefit to be obtained even when several evaluations of $y + z$ reach s .

(iii) This algorithm will miss the fact that $a * z$ and $c * z$ must have the same value in

$a := x + y$ $c := x + y$

vs

$b := a * z$ $d = c * z$

because this simple approach to common subexpressions considers only the literal expressions themselves, rather than the values computed by expressions.

Q.29. Explain following code improving transformations with examples –

(i) Local and global elimination of common subexpression.

(ii) Copy propagation and dead code elimination.

(R.G.P.V., May 2018)

Ans. (i) Local and Global Elimination of Common Subexpression – Refer to Q.4 and Q.28.

(ii) Copy Propagation and Dead Code Elimination – Refer to Q.4.

Q.30. What do you understand by copy propagation? Write an algorithm for copy propagation.

Ans. Algorithm for global common subexpression elimination, and various other algorithms such as induction-variable elimination, introduce copy statement of the form $x := y$. Copies may also be generated directly by the intermediate code generator, although most of these involve temporaries local to one block and can be removed by the DAG construction. It is sometimes possible to eliminate copy statement $s : x := y$ if we determine all places where this definition of x is used. We may then substitute y for x in these places, provided the following conditions are met by every such use u of x .

(i) Statement s must be the only definition of x reaching u .

(ii) On every path from s to u , including paths that go through u several times, there are no assignments to y .

Condition (i) can be checked using ud-chaining information, but for condition (ii), we shall set up a new data-flow analysis problem in which $in[B]$

is the set of copies $s : x := y$ such that every path from the initial node to the beginning of B contains the statement s , and subsequent to the last occurrence of s there are no assignments to y . The set $out[B]$ can be defined correspondingly, but with respect to the end of B . We say copy statement $s : x := y$ is generated in block B if s occurs in B and there is no subsequent assignment to y within B . We say $s : x := y$ is killed in B if x or y is assigned there and s is not in B .

Algorithm – Copy propagation

Input – A flow graph G , with ud-chains giving the definitions reaching block B , and with $c_in[B]$ representing the solution to equations given below, that is, the set of copies $x := y$ that reach block B along every path, with no assignment to x or y following the last occurrence of $x := y$ on the path. We also need du-chains giving the uses of each definition.

$$out[B] = c_gen[B] \cup (in[B] - c_kill[B])$$

$$in[B] = \bigcap_{P \text{ a predecessor of } B} out[P] \text{ for } B \text{ not initial}$$

$$in[B_1] = \emptyset \text{ where } B_1 \text{ is the initial block}$$

Output – A revised flow graph.

Method – For each $s : x := y$ do the following –

- (i) Determine those uses of x that are reached by this definition of x , namely, $s : x := y$.
- (ii) Determine whether for every use of x found in (i), s is in $c_in[B]$, where B is the block of this particular use; and moreover, no definitions of x or y occur prior to this use of x within B .
- (iii) If s meets the conditions of (ii), then remove s and replace all uses of x found in (i) by y .

Q.31. What is optimization? What are the common techniques for improving the intermediate code. Explain three of them in brief.

(R.G.P.V., May 2019)

Ans. Refer to Q.1, Q.4, Q.14, Q.17 and Q.30.

Q.32. What is the detection of loop-invariant computations? Write an algorithm for detection of loop-invariant computation.

Ans. The use of ud-chains to detect those computations in a loop that are loop-invariant, that is, whose value does not change as long as control stays within the loop can be made. A loop is a region consisting of a set of blocks with a header that dominates all the other blocks, so the only way to enter the loop is through the header. A loop should also have at least one way to get back to the header from any block in the loop.

If an assignment $x := y + z$ is at a position in the loop where all possible definitions of y and z are outside the loop, then $y + z$ is loop-invariant because its value will be the same each time $x = y + z$ is encountered, as long as control stays within the loop.

Having recognized that the value of x computed at $x := y + z$ does not change within the loop, suppose there is another statement $v := x + w$, where w could only have been defined outside the loop. Then $x + w$ is also loop-invariant.

The above ideas can be used to make repeated passes over the loop, discovering a large number of computations whose value is loop-invariant. If we have both ud-and du-chains, we do not even have to make repeated passes over the code. The du-chain for definition $x := y + z$ will tell us where this value of x could be used, and we need only check among these uses of x , within the loop, that use no other definition of x .

Algorithm – Detection of loop-invariant computations.

Input – A loop L consisting of a set of basic blocks, each block containing a sequence of three-address statements. We assume ud-chains, as computed in introduction to global data-flow analysis, are available for the individual statements.

Output – The set of three-address statements that compute the same value each time executed, from the time control enters the loop L until the control next leaves L .

Method – We shall give a rather informal specification of the algorithm, trusting that the principles will be clear.

- (i) Mark “invariant” those statements whose operands are all either constant or have all their reaching definitions outside L .
- (ii) Repeat step (iii) until at some repetition no new statements are marked “invariant”.
- (iii) Mark “invariant” all those statements not previously so marked all of whose operands either are constant, have all their reaching definitions outside L , or have exactly one reaching definition, and that definition is a statement in L marked invariant.

Q.33. Explain the common subexpression elimination, copy propagation and transformation for moving loop invariant computations in detail.

(R.G.P.V., Dec. 2017)

Ans. Refer to Q.4, Q.30 and Q.32.

Q.34. What is code motion? Write an algorithm for code motion.

Ans. Having found the invariant statements within a loop, we can apply to some of them an optimization known as *code motion*, in which the statements are moved to the preheader of the loop. The following three conditions ensure that code motion does not change what the program computes. None of the

condition is so essential, we have selected these conditions because they are easy to check and apply to situations that occur in real programs.

The conditions for statement $s : x := y + z$ are –

(i) The block containing s dominates all exit nodes of the loop, where an exit of a loop is a node with a successor not in the loop.

(ii) There is no other statement in the loop that assigns to x . Again, if x is a temporary assigned only once, this condition is surely satisfied and need not be checked.

(iii) No use of x in the loop is reached by any definition of x other than s . This condition too will be satisfied, normally, if x is a temporary.

Algorithm – Code motion.

Input – A loop L with ud-chaining information and dominator information.

Output – A revised version of the loop with a preheader and some statements moved to the preheader.

Method – (i) Use algorithm for detection of loop-invariant computation, to find loop-invariant statements.

(ii) For each statement s defining x found in step (i), check –

(a) That it is in a block that dominates all exits of L ,

(b) That x is not defined elsewhere in L and

(c) That all uses in L of x can only be reached by the definition of x in statement s .

(iii) Move, in order found by algorithm for detection of loop-invariant computation, each statement s found in (i) and meeting conditions (ii) (a), (ii) (b) and (ii) (c), to a newly created preheader, provided any operands of s that are defined in loop L have previously had their definition statements moved to the preheader.

Q.35. What are the induction variables? Write an algorithm for detection of induction variables.

Ans. A variable x is called an *induction variable* of a loop L if every time the variable x changes values, it is incremented or decremented by some constant. Often, an induction variable is incremented by the same constant each time around the loop, as i in a loop headed by **for** $i = 1$ **to** 10. The number of changes to an induction variable may even differ at different iterations.

A common situation is one in which an induction variable, say i , indexes an array and some other induction variable, say t , whose value is a linear function of i , is the actual offset used to access the array. Often, the only use made of i is in the test for loop termination. We can then get rid of i by replacing its test by one on t .

We shall look for *basic induction variables*, which are those variables i whose only assignments within loop L are of the form $i := i \pm c$, where c is a constant. We then look for additional induction variables j that are defined only

once within L , and whose value is a linear function of some basic induction variable i where j is defined.

Algorithm – Detection of induction variable.

Input – A loop L with reaching definition information and loop-invariant computation information.

Output – A set of induction variables. Associated with each induction variable j is a triple (i, c, d) where i is a basic induction variable, and c and d are constants such that the value of j is given by $c*i + d$ at the point where j is defined. We say that j belongs to the *family* of i . The basic induction variable i belongs to its own family.

Method –

(i) Find all basic induction variables by scanning the statements of L . We use the loop-invariant computation information here. Associated with each basic induction variable i is the triple $(i, 1, 0)$.

(ii) Search for variables k with a single assignment to k within L having one of the following forms –

$$k := j * b, k := b * j, k := j / b, k := j \pm b, k := b \pm j$$

where b is a constant and j is an induction variable, basic or otherwise.

If j is basic, then k is in the family of j . The triple for k depends on the instruction defining it. For example, if k is defined by $k := j * b$, then the triple for k is $(j, b, 0)$. The triples for the remaining cases can be determined similarly.

If j is not basic, let j be in the family of i . Then our additional requirements are that –

(a) There is no assignment to i between the lone point of assignment to j in L and the assignment to k , and

(b) No definition of j outside L reaches k .

The usual case will be where the definitions of k and j are in temporaries in the same block, in which case it is easy to check. Generally, reaching definitions information will provide the check we need if we analyze the flow graph of loop L to determine those blocks on paths between the assignment to j and the assignment to k .

We compute the triple for k from the triple (i, c, d) for j and the instruction defining k . For example, the definition $k := b * j$ leads to $(i, b * c, b * d)$ for k .

Q.36. What are induction variables? How does the strength reduction on induction variable help in improving loop optimization? Illustrate with an example.

(R.G.P.V., Dec. 2015)

Ans. Refer to Q.35.

Assume the input source and its unoptimized intermediate code as shown in fig. 5.7.

Input Source	TAC
1. int temp;	(0) proc_begin func
2. int x [15]	(1) label . L0
3.	(2) if temp < 15 goto.L1
4. int func ()	(3) goto.L2
5. {	(4) label.L1
6. while (temp < 15)	(5) $_t0 := temp * 4$
7. {	(6) $_t1 := \& x$
8. x [temp] = 20;	(7) $_t1 [_t0] := 20$
9. temp = temp + 2;	(8) temp := temp + 1
10. }	(9) goto.L0
11. }	(10) label.L2
	(11) label.L3
	(12) proc_end func

Fig. 5.7

In fig. 5.7, temp variable is a user defined induction variable increased by 2 on each iteration. Variable $_t0$ is another induction variable generated by the compiler and increased by 4 on each iteration. The intermediate code after the application of reduction of strength on the induction variable $_t0$ is shown in fig. 5.8.

In fig. 5.8, the initial value of $_t0$, i.e. $_t0 := temp * 4$ is moved out of the loop. This is used as the initial value, for $_t0$. An additional computing the value of $_t0$ from its previous value i.e. $_t0 := _t0 + 4$ is inserted, which computes the value of main induction variable temp. The transformed loop in fig. 5.8 is functionally equivalent to the intermediate code in fig. 5.7. The strength reduction transformations are usually applied on the induction variables in a loop to get substantial performance benefits. The strength reduction transformation on induction variables improves the speed of execution because we have less expensive instruction.

(0) proc_begin func
(1) $_t0 := temp * 4$
(2) label.L0
(3) if temp < 15 goto.L2
(4) goto.L3
(5) label.L2
(6)
(7) $_t1 := \& x$
(8) $_t1 [_t0] := 20$
(9) temp := temp + 2
(10) $_t0 := _t0 + 4$
(11) goto.L0
(12) label.L3
(13) label.L4
(14) proc_end func

Fig. 5.8

Q.37. Write an algorithm for elimination of induction variables.

Or

Write short note on induction variable elimination. (R.G.P.V., May 2018)

Ans. Algorithm – Elimination of induction variables.

Input – A loop L with reaching definition information loop-invariant computation information and live variable information.

Output – A revised loop.

Method – (i) Consider each basic induction variable i whose only uses are to compute other induction variables in its family and in conditional branches. Take some j in i 's family, preferably one such that c and d in its triple (i, c, d) are as simple as possible, and modify each test i that appears in to use j instead. We assume in the following that c is positive. A test of the form $\text{if } i \text{ relop } x \text{ goto } B$, where x is not an induction variable, is replaced by

$r := c * x$ $/* r := x \text{ if } c \text{ is } 1 */$
 $r := r + d$ $/* omit if } d \text{ is } 0 */$
 if j relop r goto B

where r is a new temporary. The case $\text{if } x \text{ relop } i \text{ goto } B$ is handled analogously. If there are two induction variables i_1 and i_2 in the test $\text{if } i_1 \text{ relop } i_2 \text{ goto } B$, then we check if both i_1 and i_2 can be replaced. The easy case is when we have j_1 with triple (i_1, c_1, d_1) and j_2 with triple (i_2, c_2, d_2) , and $c_1 = c_2$ and $d_1 = d_2$. Then, $i_1 \text{ relop } i_2$ is equivalent to $j_1 \text{ relop } j_2$. In more complex cases, replacement of the test may not be worthwhile, because we may need to introduce two multiplicative steps and one addition, while as few as two steps may be saved by eliminating i_1 and i_2 .

Finally delete all assignments to the eliminated induction variables from the loop L , as they will now be useless.

(ii) Now, consider each induction variable j for which a statement $j := s$ was introduced by algorithm for strength reduction applied to induction variables. First check that there can be no assignment to s between the introduced statement $j := s$ and any use of j . In the usual situation, j is used in the block in which it is defined, simplifying this check; otherwise, reaching definitions information, plus some graph analysis is needed to implement the check. Then replace all uses of j by uses of s and delete statement $j := s$.

Q.38. Write an algorithm for strength reduction applied to induction variables.

Ans. Algorithm – Strength reduction applied to induction variables.

Input – A loop L with reaching definition information and families of induction variables computed using algorithm for detection of induction variables.

Output – A revised loop.

Method – Consider each basic induction variable i in turn. For every induction variable j in the family of i with triple (i, c, d) –

(i) Create a new variable s (but if two variable j_1 and j_2 have the same triples, just create one new variable for both).

(ii) Replace the assignment to j by $j := s$.

(iii) Immediately after each assignment $i := i + n$ in L , where n is a constant, append –

$s := s + c * n$

where the expression $c * n$ is evaluated to a constant because c and n are constants. Place s in the family of i , with triple (i, c, d) .

(iv) It remains to ensure that s is initialized to $c*i + d$ on entry to the loop. The initialization may be placed at the end of the preheader. The initialization consists of

```

s : c*i      /*just s := i if c is 1*/
s := s + d   /*omit if d is 0*/

```

Q.39. Discuss the different edges in depth-first presentation of flow graph.

Ans. When we construct a *dfst* for a flow graph, the edges of the flow graph fall into three categories as follows –

(i) There are edges that go from a node m to an ancestor of m in the tree. These edges are termed as retreating edges. It is an interesting and useful fact that if the flow graph is reducible, then the retreating edges are exactly the back edges of the flow graph, independent of the order in which successors are visited.

(ii) The edges that go from a node m to a proper descendant of m in the tree are called *advancing edges*. All edges in the *dfst* itself are advancing edges.

(iii) The edges $m \rightarrow n$ such that neither m nor n is an ancestor of the other in the *dfst* are called *cross edges*. An important property of *cross edges* is that if we draw the *dfst* so children of a node are drawn from left to right in the order in which they were added to the tree, then all cross edges travel from right to left.

Q.40. What is symbolic debugger ?

Ans. A system that allows us to look at a program's data while the program is running is known as *symbolic debugger*. A debugger is usually called when a program error, such as an overflow, occurs or when certain statements, indicated by the programmer in the source code, are reached. Once invoked, the symbolic debugger allows the programmer to examine, and possibly change, any of the variables that are currently accessible to the running program.

Q.41. What kind of information a symbolic debugger must have ?

Ans. In order for a user command like “show me the current value of a ”, to be intelligible to the debugger, it must have available certain information such as –

(i) There must be a way to associate an identifier like a with the location it represents. Thus, the portion of the symbol table that assigns to each variable a location, e.g., a place in a global data area or in an activation record for some procedure, must be recorded by the compiler and preserved for the debugger to use.

(ii) There must be scope information, so we can disambiguate references to an identifier that is declared more than once, and so that we can

tell, given we are in some procedure p , what other procedure's data is accessible and how do we find that data on the stack or other run-time structure.

(iii) We must know where we are in the program when the debugger is invoked. This information is embedded by the compiler in the call to the debugger when the compiler handles a user-declared invocation of the debugger. It is also obtained from the exception handler when run-time error causes the debugger to be called.

(iv) In order that program-location information, mentioned in (iii), make sense to the user, there must be a table associating each machine language statement with the source statement from which it came. This table can be prepared by the compiler as it generates code.

Q.42. What do you mean by symbolic debugging of optimized code ?

(R.G.P.V., June 2016)

Ans. Refer to Q.40 and Q.41.

Q.43. How can we deduce the values of variables in basic blocks ?

Ans. We assume that both the source and object code are sequences of intermediate statements. Treating the source as intermediate code presents no problems, since the latter is more general than former. For example, the user may only be allowed to put breaks between source statements, but here we allow breaks after any intermediate statement. Treating the object code as intermediate code is questionable only if the optimizer breaks a single intermediate statement into several machine statements that get separated. For example, for some reason, we may compile the two intermediate statements

```

u := v + w
x := y + z

```

into code where the two additions are performed in different registers and interleaved. If that is the case, we can treat the loads and stores of registers as if the registers were temporaries in intermediate code, for example,

```

r1 := v
r2 := y
r1 := r1 + w
r2 := r2 + z
u := r1
x := r2

```

Several problems occur when interacting with the user about a block, where the user thinks the source block is being executed, but in fact, an optimized version of that block is running –

(i) Suppose we are executing the program that results from “optimizing” some basic block of the source program, and while executing statement $a := b + c$, an overflow occurs. We must tell the user that an error has occurred in one of the source statements. Since $b + c$ may be a common

subexpression appearing in two or more of the source statements, to which statement do we charge the error?

(ii) A harder problem occurs if the user of the debugger wants to see the "current" value of some variable d . In the optimized program, d may last have been assigned at some statement s . But in the source program, s may come after the statement at which the debugger was invoked, so the value of d that is available to debugger is not the one that the user thinks is the "current" value of d according to the listing of the source code. Similarly s may precede the statement invoking the debugger, but in the source there is another assignment to d between them, so the value of d available to the debugger is out of date. Is it possible to make the correct value of d available to the user? For example, could it be the value of some other variable in the optimized version, or could it be computed from the values of other variables?

(iii) Finally, if the user places a break after some statement of the source code, when should control be given to the debugger during the running of the optimized code?

One solution might be to run the unoptimized version of the block along with the optimized version, to make the correct value of each variable available at all times. We reject this "solution" because the subtlest of bugs, especially compiler-introduced bugs, may disappear when the instructions that caused the problem are separated from one another in time or space.



B.E. (Seventh Semester) EXAMINATION, Dec. 2010
(New Scheme)
(Computer Science & Engg. Branch)
COMPILER DESIGN
[CS-701(N)]

RGPV

Note : Attempt all questions. All questions carry equal marks.

1. (a) What are the tasks performed by the compiler in the lexical and syntax analysis phases? (See Unit-I, Page 20, Q.22) 10
- (b) What is LEX? Describe auxiliary definitions and translation rules of LEX with suitable example. (See Unit-I, Page 32, Q.39) 10

Or

2. (a) Compare and contrast the features of a single pass compiler with multipass compiler. (See Unit-I, Page 6, Q.6) 10
- (b) Describe the role of a lexical analyzer and also explain the concept of input buffering. (See Unit-I, Page 27, Q.34) 10
3. (a) What is top down parsing? What are the difficulties encountered in this and how are they overcome? (See Unit-II, Page 47, Q.7) 10
- (b) Design LL(1) parsing table for the following grammar: 10

$A \rightarrow AcB|cC|C$

$B \rightarrow bB|id$

$C \rightarrow CaB|BbB|B$

(See Unit-II, Page 68, Prob.13)

Or

4. (a) For the following grammar find FIRST and FOLLOW sets for each of non terminal: 10
- $S \rightarrow aAB|bA|\epsilon$
 $A \rightarrow aAb|\epsilon$
 $B \rightarrow bB|\epsilon$

(See Unit-II, Page 67, Prob.11)

- (b) (i) What is a predictive parser? How a parser is controlled by a program? (See Unit-II, Page 57, Q.15) 6

(ii) Write a short note on YACC. (See Unit-II, Page 92, Q.43) 4

5. (a) Differentiate between implicit type conversion and explicit type conversion with the help of an example. (See Unit-III, Page 136, Q.11) 10
- (b) What is activation record? Give the general activation record fields and their purpose. (See Unit-III, Page 143, Q.20) 10

Or

6. (a) What is hashing? What are different types of hashing techniques available? (See Unit-III, Page 165, Q.41) 10
- (b) (i) Describe parameter passing mechanisms for a procedure call. 6
 (See Unit-III, Page 152, Q.29)
- (ii) Write a short note on symbol table organization. 4
 (See Unit-III, Page 159, Q.35)

(1)

7. (a) Translate the expression :
 $-(a+b)*(c+d) - (a+b+c)$
 into quadruples, triples and indirect triples.
 (See Unit-IV, Page 199, Prob.13) 10
- (b) What are the advantages of DAG ? Describe the process of code generation from DAG.
 (See Unit-IV, Page 227, Q.48) 10
- Or
8. (a) Construct DAG of basic blocks after converting the code in 3-address representation :
 $i = 1;$
 $j = 2;$
 Repeat: $A[i] = j$
 $j = j * 2;$
 $i = i + 1;$
 Until $(i > 10)$
 (b) Construct 3 address code for the following :
 if: $[(a < b) \text{ and } ((c > d) \text{ or } (a > d))]$
 then :
 $z = x + y * z$
 else :
 $z = z + 1$
 (See Unit-IV, Page 193, Prob.2) 10
9. (a) Explain the principle sources of optimization with suitable example.
 (See Unit-V, Page 233, Q.4) 10
- (b) Explain the Peephole optimization.
 (See Unit-IV, Page 221, Q.46) 10
- Or
10. (a) What is global data flow analysis ? What is its use in code optimization?
 (See Unit-V, Page 243, Q.25) 10
- (b) What is a basic block ? Discuss various transformations that can be on basic block with the help of suitable example.
 (See Unit-IV, Page 210, Q.33) 10

RGPV

B.E. (Seventh Semester) EXAMINATION, June 2011
(Computer Science & Engg. Branch)
COMPILER DESIGN
[CS-701(N)]

Note : Attempt all questions. All questions carry equal marks.

1. (a) Describe the common data structures used by a compiler.
 (See Unit-I, Page 4, Q.2) 10
- (b) Discuss the different phases in a compiler ? Explain each one of them.
 (See Unit-I, Page 15, Q.20) 10
- Or
2. (a) What are the various components of a lexical specification file ? Illustrate with an example.
 (See Unit-I, Page 35, Q.40) 10

(2)

- (b) (i) What is a 'pass' in a compiler ? Differentiate between multiple pass compiler and a single pass compiler ?
 (See Unit-I, Page 5, Q.5) 5
- (ii) Explain the process of bootstrapping used in compiler.
 (See Unit-I, Page 7, Q.10) 5
3. (a) What are the different methods of constructing the LR parsing table from a given grammar ?
 (See Unit-II, Page 90, Q.41) 10
- (b) Show that the given grammar is LL(1) :
 $S \rightarrow AaAb \mid BbBa$
 $A \rightarrow \epsilon$
 $B \rightarrow \epsilon$
 (See Unit-II, Page 69, Prob.14) 10
- Or
4. (a) Consider the grammar :
 $S \rightarrow ACB \mid CbB \mid Ba$
 $A \rightarrow da \mid BC$
 $B \rightarrow g \mid \epsilon$
 $C \rightarrow h \mid \epsilon$
 Calculate FIRST and FOLLOW.
 (See Unit-II, Page 66, Prob.9) 10
- (b) (i) Differentiate between synthesized translation and inherited translation.
 (See Unit-II, Page 116, Q.51) 6
- (ii) Write a short note on syntax directed translation.
 (See Unit-II, Page 110, Q.44) 4
5. (a) Explain : implicit type conversion and explicit type conversion.
 (See Unit-III, Page 136, Q.11) 10
- (b) What is the difference between dynamic and static storage management ? Explain the importance of run time storage management in compiler.
 (See Unit-III, Page 152, Q.28) 10
- Or
6. Write short notes on the following -
 (i) Activation record
 (See Unit-III, Page 143, Q.20) 20
- (ii) Symbol table
 (See Unit-I, Page 5, Q.3) 20
- (iii) Symbol table organization
 (See Unit-III, Page 159, Q.35) 20
- (iv) Parameter passing
 (See Unit-III, Page 152, Q.29) 20
7. (a) Generate the three address code for the following program fragment -
 while $(A < C \text{ and } B > D)$ do
 if $A = 1$ then $C = C + 1$
 else while $A \leq D$ do
 $A = A + 3$
 (See Unit-IV, Page 194, Prob.3) 10
- (b) Explain heuristic addressing algorithm for DAG.
 (See Unit-IV, Page 218, Q.42) 10
- Or
8. (a) Explain the various applications of DAG. Construct DAG for the following basic block -
 $D := B * C$
 (3) 10

$E := A + B$
 $B := B * C$
 $A := E - D$

- (b) Compare the relative merits and demerits of the following – (See Unit-IV, Page 218, Q.41) 10

- (i) Quadruples
 (ii) Triples
 (iii) Indirect triples

and write the quadruples, triple and indirect triples for the expression –
 $(a * b) + (c + d) - (a + b + c + d)$

9. (a) What is global data flow analysis ? What is its use in code optimizations ? (See Unit-IV, Page 174, Q.9) 10

- (b) Describe the necessary and sufficient conditions for performing constant propagation and dead code elimination. (See Unit-V, Page 243, Q.25) 10

Or

10. (a) Explain the following optimizing transformation with suitable examples – (See Unit-V, Page 237, Q.13) 10

- (i) Code movement
 (ii) Common subexpression elimination.

- (b) Explain briefly the following – (See Unit-V, Page 236, Q.8) 10

- (i) Peephole optimization (See Unit-IV, Page 221, Q.46) 10
 (ii) Data flow equations. (See Unit-V, Page 243, Q.25)

RGPV

B.E. (Seventh Semester)
EXAMINATION, Dec. 2011
(Computer Science & Engg. Branch)
COMPILER DESIGN
(CS-701)

Note : Attempt all questions. Internal choice is given. All questions carry equal marks.

Unit-I

1. (a) Discuss the various tasks performed by the compiler in the lexical and syntax analysis phase. (See Unit-I, Page 20, Q.22) 10

- (b) Write short notes on the following : (See Unit-I, Page 7, Q.10) 10

- (i) Bootstrapping
 (ii) Compiler writing tools

Or

2. (a) What is LEX ? Describe auxiliary definitions and translation rules for LEX with suitable examples. (See Unit-I, Page 32, Q.39) 10

- (b) Discuss the advantages and disadvantages of single pass and multipass

(4)

(See Unit-I, Page 6, Q.8) 10

compilers.

Unit-II

3. (a) Consider the following grammar :

$A \rightarrow ABd \mid Aa \mid a$
 $B \rightarrow Be \mid b$

Remove left recursion from above grammar. (See Unit-II, Page 64, Prob.5) 10

- (b) What do you understand by TOP down translation ? Explain taking any example of your choice. (See Unit-II, Page 121, Q.58) 10

Or

4. (a) Draw the syntax tree and DAG for the expression : (See Unit-IV, Page 230, Prob.24) 10

$(a * b) + (c - d) * (a * b) + b$

- (b) Consider the grammar :

$S \rightarrow ACB \mid CbB \mid Ba$
 $A \rightarrow da \mid BC$
 $B \rightarrow g \mid E$
 $C \rightarrow h \mid E$

Calculate FIRST and FOLLOW.

(See Unit-II, Page 66, Prob.9) 10

Unit-III

5. (a) Differentiate between Implicit type conversion and Explicit type conversions with the help of an example. (See Unit-III, Page 136, Q.11) 10

- (b) Explain various storage allocation strategies. Which storage allocation model is to be used if a language permits recursion ? (See Unit-III, Page 145, Q.22) 10

Or

6. (a) What do you mean by heap allocation ? Explain the various terms related to heap allocation : (See Unit-III, Page 151, Q.27) 10

- (i) Free list (See Unit-III, Page 151, Q.27)
 (ii) Reference count (See Unit-III, Page 151, Q.27)
 (iii) Fragmentation (See Unit-III, Page 151, Q.27)
 (iv) Bit map

- (b) Explain the various data structures used for implementing the symbol table and compare them. (See Unit-III, Page 165, Q.40) 10

Unit-IV

7. (a) Construct 3 address code for the following : (See Unit-IV, Page 193, Prob.2) 10

if $[(a < b) \text{ and } ((c > d) \text{ or } (a > d))]$ then
 $z = x + y * z$
 else $z = z + 1$

- (b) Construct the DAG for the following basic block :

(5)

d := b * c
e := a + b
b := b * c
a := e - d

Then generate the code for the above constructed DAG using only one register. (See Unit-IV, Page 228, Prob.19) 10

Or

8. (a) Using Backpatching, generate an intermediate code for the following expression : 10

A < B OR C < D AND P < Q (See Unit-IV, Page 202, Prob.15)

(b) Write quadruples, triples and indirect triples for the expression : 10
- (a + b) * (c + d) - (a + b + c) (See Unit-IV, Page 199, Prob.13)

Unit-V

9. (a) Describe the necessary and sufficient conditions for performing constant propagation and dead code eliminations. 10

(See Unit-V, Page 237, Q.13)

(b) What is global data flow analysis ? What is in use in code optimization ? (See Unit-V, Page 243, Q.25) 10

Or

10. Write short notes on the following : 20

- (i) Peephole optimization (See Unit-IV, Page 221, Q.46)
- (ii) Code movement (See Unit-V, Page 236, Q.8)
- (iii) Common subexpression elimination (See Unit-V, Page 236, Q.8)
- (iv) Data flow equations. (See Unit-V, Page 243, Q.25)

RGPV

B.E. (Seventh Semester)
EXAMINATION, June 2012
(Computer Science & Engg. Branch)
COMPILER DESIGN
(CS-701)

Note : Attempt one question (including all parts) from each Unit. All questions carry equal marks. Assume missing data, if any suitably.

Unit-I

1. (a) What is a Translator ? Compare Compiler, Assembler and Interpreter. (See Unit-I, Page 11, Q.15)
- (b) Discuss the advantages and disadvantages for single and multipass compilers. (See Unit-I, Page 6, Q.8)

Or

2. (a) What is Lex ? Describe auxiliary definitions and translation rules for LEX with suitable example. (See Unit-I, Page 32, Q.39)
- (b) What is a cross compiler ? Create a cross compiler C_s^{LA} using previously known language C_s^{SA} . (See Unit-I, Page 10, Q.13)

Unit-II

3. (a) Construct LALR items for the grammar below :

$B \rightarrow E + T | T$
 $T \rightarrow T * F | F$
 $F \rightarrow (E) | id$

(See Unit-II, Page 102, Prob.26)

(b) Find LEADING and TRAILING of the following grammars :

(i) $S \rightarrow aAB|bA|\epsilon$
 $A \rightarrow aAb|\epsilon$
 $B \rightarrow bB|c$

(See Unit-II, Page 78, Prob.18)

(ii) $S \rightarrow b|DAC|AB,aCB$
 $A \rightarrow c|CbD|SABC|SCB|\epsilon$
 $B \rightarrow d|cD|\epsilon$
 $C \rightarrow c|ADC$
 $D \rightarrow fg|SaC|SC$

Or

4. (a) What is meant by syntax directed translation ? Explain. Give the parse tree and translations for the expression $25*5 + 4$ according to the syntax directed translation scheme. (See Unit-II, Page 114, Q.47)

(b) Consider the following grammar :

$S \rightarrow 1AB|\epsilon$
 $A \rightarrow 1AC|OC$
 $B \rightarrow OS$
 $C \rightarrow 1$

and test whether the grammar is LL(1) or not.

(See Unit-II, Page 71, Prob.16)

Unit-III

5. (a) Explain the various data structures used for implementing the symbol table and compare them. (See Unit-III, Page 165, Q.40)
- (b) What is activation Record ? Give the general activation record field and their purpose. (See Unit-III, Page 143, Q.20)

Or

6. (a) Explain the importance of run-time storage management in compilers. (See Unit-III, Page 142, Q.18)
- (b) Describe the checking rules for polymorphic functions. (See Unit-III, Page 139, Q.15)

Unit-IV

7. (a) What are the general issues in designing a Code Generator ? (See Unit-IV, Page 205, Q.29)
- (b) What is DAG ? What are its advantages in context of Optimization ? How does it help in elimination of common sub expressions ? (See Unit-IV, Page 220, Q.45)

Or

8. (a) What are the problems encountered in code generation ? (See Unit-IV, Page 204, Q.28)

- (b) Show the DAG for the following statement :

$$Z = X - Y + X * Y * U - V / W + X + V$$

(See Unit-IV, Page 229, Prob.23)

Unit-V

9. (a) Translate the expression :

$$A := -B * (c + d) / E$$

in to quadruples and triples representations.

(See Unit-IV, Page 198, Prob.10)

- (b) What is global data flow analysis ? What is its use in code optimization ?

(See Unit-V, Page 243, Q.25)

10. (a) Explain the need of code optimization. With example, illustrate loop optimization.

(See Unit-V, Page 237, Q.11)

- (b) Explain the backpatching. Also generate three-address code for the following program segment :

while (a < c and b > d) do

if a = 1 then c = c + 1;

else

while a <= d do

a = a + 3;

(See Unit-IV, Page 194, Prob.4)

RGPV

**B.E. (Seventh Semester)
EXAMINATION, Dec. 2012
(Computer Science & Engg. Branch)
COMPILER DESIGN
(CS-701)**

- Note :** (i) Attempt all questions. All questions carry equal marks.
(ii) Each question have internal choice.

Unit-I

1. (a) What do you understand by automatic lexical generator ? 10

(See Unit-I, Page 23, Q.29)

- (b) Explain various phases of a compiler. 10

(See Unit-I, Page 15, Q.20)

Or

2. (a) Write a LEX specification file to identify the tokens of the language C.

(See Unit-I, Page 35, Q.41)

- (b) Construct FA for the regular expressions

(i) $(a + b)^*abb$ (ii) $((a^* + b)^* + b^*)^*$

(See Unit-I, Page 39, Prob.5) 10

Unit-II

3. (a) Describe the error reporting and recovery schemes in operator precedence parsing. 10

(See Unit-II, Page 77, Q.33)

- (b) Explain S-attribute and L-attribute.

(See Unit-II, Page 121, Q.56) 10

Or

4. For the following grammar –

(8)

$$E \rightarrow E + T / T$$

$$T \rightarrow T * F / F$$

$$F \rightarrow (E) / id$$

Construct the LR(0) canonical collection and also design the SLR parsing table. (See Unit-II, Page 94, Prob.21) 20

Unit-III

5. (a) Discuss the importance of type equivalence checking. 10

(See Unit-III, Page 134, Q.6)

- (b) Discuss the importance of symbol table in compiler design. How is the symbol table manipulated at various phases of compilation ? 10

(See Unit-I, Page 20, Q.21)

Or

6. (a) Discuss the following storage-allocation strategies – 10

(i) Stack allocation

(ii) Heap allocation.

(See Unit-III, Page 150, Q.25)

- (b) Compare explicit and implicit type conversion. 10

(See Unit-III, Page 136, Q.11)

Unit-IV

7. (a) Write triples for the expression

$$(a + b) * (c + d) - (a + b + c)$$

(See Unit-IV, Page 199, Prob.12)

- (b) Show the annotated parse tree and code generation process for the arithmetic expression $a + (b - c) + d$. (See Unit-IV, Page 215, Prob.16) 10

Or

8. (a) Construct the DAG for 10

$$X = Y * Z$$

$$W = P + Y$$

$$Y = Y * Z$$

$$P = W - X$$

(See Unit-IV, Page 227, Prob.18)

- (b) Discuss the factors affecting target code generation. 10

(See Unit-IV, Page 204, Q.28)

Unit-V

9. (a) Explain different type of optimization. (See Unit-V, Page 239, Q.18) 10

- (b) What are the common algebraic transformations that can be done for improving the intermediate code? (See Unit-V, Page 239, Q.17) 10

Or

10. (a) Explain dead code elimination with example. 10

(See Unit-V, Page 236, Q.7)

- (b) What is global data flow analysis ? What is its use in code optimization? (See Unit-V, Page 243, Q.25) 10

Note : Attempt one question from each unit.

Unit-I

- Explain the role of the lexical analysis in compiler and also discuss the issues in lexical analysis. (See Unit-I, Page 25, Q.32)
 - Construct the NFA and then optimized DFA for the regular expression $(a/b)^*a^*$
- Explain the following –
 - Boot strapping and Porting (See Unit-I, Page 10, Q.14)
 - Input buffering (See Unit-I, Page 25, Q.33)
 - Describe the analysis-synthesis model of compilation. (See Unit-I, Page 13, Q.18)

Unit-II

- What are the merits and demerits of LR-parser ? Construct SLR parsing table for the following grammar with necessary codes for action

$$E \rightarrow E + T \mid T$$

$$T \rightarrow T * F \mid F$$

$$F \rightarrow (E) \mid id$$
 (See Unit-II, Page 84, Q.36)
- Explain the syntax directed definition for constructing syntax tree for an arithmetic expression. Also explain what is annotated parse tree. (See Unit-II, Page 110, Q.44)
 - Define context free grammar and explain how it is suitable for parsing. (See Unit-II, Page 43, Q.3)

Unit-III

- Explain the memory allocation in block structured languages. (See Unit-III, Page 142, Q.19)
 - Distinguish between static scope and dynamic scope. Briefly explain access to non-local names in static scope. (See Unit-III, Page 155, Q.30)
- Explain in detail different dynamic storage allocation strategies. (See Unit-III, Page 157, Q.33)
 - What is type conversion ? (See Unit-III, Page 135, Q.9)
 - Explain overloading of functions and operations briefly. (See Unit-III, Page 137, Q.12)

Unit-IV

- Discuss the issues in the design of code generator. (See Unit-IV, Page 205, Q.29)

- Give the translation scheme for converting the assignments into three address code. (See Unit-IV, Page 176, Q.11)
- Explain the code generation algorithm and generate the code for the following –

$$X = (A - B) + (A - C) * (A - C)$$
 (See Unit-IV, Page 209, Q.32)
 - Describe peephole optimization briefly. (See Unit-IV, Page 221, Q.46)

Unit-V

- What is code optimization ? How is it achieved ? Explain. (See Unit-V, Page 231, Q.1)
 - Explain data flow analysis of structure flow graph. (See Unit-V, Page 243, Q.25)
- Write short notes on the following –
 - Loops in flow graphs (See Unit-V, Page 241, Q.22)
 - Loop unrolling, peeling, fusion (See Unit-V, Page 242, Q.24)
 - Code improving transformations (See Unit-V, Page 246, Q.26)
 - Function preserving transformations. (See Unit-V, Page 236, Q.5)

Note : (i) Attempt one question from each unit.
(ii) All questions carry equal marks.

Unit-I

- Explain the various phases of compiler ? How phases of compilation converts the statement

$$\text{Position} = \text{initial} + \text{rate} * 60$$
 (See Unit-I, Page 15, Q.20)
 - Briefly explain the compiler construction tools. (See Unit-I, Page 20, Q.24)
- What are the issues in lexical analysis ? Explain in detail the recognition of tokens. (See Unit-I, Page 31, Q.37)
 - Design FA to accept the following –
 - Identifiers
 - Constant.
 (See Unit-I, Page 38, Prob.1)

Unit-II

- Explain handle pruning. Explain the same for the grammar

$$E \rightarrow E + E / E * E / (E) / id$$
 and input string $id1 + id2 * id3$. (See Unit-II, Page 63, Q.24)
 - Describe the conflicts that may occur during shift reduce parsing. (See Unit-II, Page 61, Q.20)
- Check whether the given grammar is LL(1) or not.

$S \rightarrow iEtSS'/a$

$S' \rightarrow eS/E$

$E \rightarrow b$

(See Unit-II, Page 70, Prob.15)

(b) What is syntax directed translation? Why are they important?

(See Unit-II, Page 110, Q.44)

Unit-III

5. (a) Discuss the symbol table organization, also give the difference between binary tree and hashing organization of symbol table.

(See Unit-III, Page 164, Q.39)

(b) Explain the various parameter passing mechanism.

(See Unit-III, Page 152, Q.29)

6. (a) Explain the specification of simple type checker.

(See Unit-III, Page 130, Q.3)

(b) What is polymorphic functions?

(See Unit-III, Page 138, Q.13)

(c) How type checking and type conversion is implemented in compiler?

(See Unit-III, Page 136, Q.10)

Unit-IV

7. (a) Construct DAG for the following expression

$a + a * (b - c) + (b - c) * d$ (See Unit-IV, Page 229, Prob.22)

(b) How CPU registers are allocated while creating machine code?

(See Unit-IV, Page 213, Q.37)

8. (a) Write quadruples from the expression –

$(a + b) * (c + d) - (a + b + c)$ (See Unit-IV, Page 198, Prob.11)

(b) Discuss the issues in design of code generator.

(See Unit-IV, Page 205, Q.29)

Unit-V

9. (a) What is global data analysis? What is its use in code optimization?

(See Unit-V, Page 243, Q.25)

(b) Explain the following with example –

(i) Strength reduction

(ii) Variable propagation

(iii) Common subexpression elimination.

(See Unit-V, Page 237, Q.12)

10. (a) Explain loop optimization with example. (See Unit-V, Page 236, Q.10)

(b) Define the following –

(i) Dominators (ii) Natural loops

(iii) Inner loop (iv) Preheader.

(See Unit-V, Page 242, Q.23)

RGPV

B.E. (Seventh Semester)
EXAMINATION, Dec. 2015
(Computer Science & Engg. Branch)
COMPILER DESIGN
(CS-701)

Note : (i) Answer five questions. In each question part A, B, C is compulsory and D part has internal choice.

(ii) All parts of each questions are to be attempted at one place.

(iii) All questions carry equal marks, out of which part A and B (Max. 50 words) carry 2 marks, part C (Max. 100 words) carry 3 marks, part D (Max. 400 words) carry 7 marks.

(iv) Except numericals, Derivation, Design and Drawing etc.

Unit-I

1. (a) Write the advantage of multipass compiler over single pass compiler.

(See Unit-I, Page 6, Q.7)

(b) What is bootstrapping? Also explain cross compiler.

(See Unit-I, Page 8, Q.12)

(c) What are the various components of a lexical specification file? Illustrate with an example.

(See Unit-I, Page 35, Q.40)

(d) What is LEX? Describe auxiliary definitions and translation rules for LEX with suitable example.

(See Unit-I, Page 32, Q.39)

Or

Discuss the various phases of compiler with the help of neat labelled diagram.

(See Unit-I, Page 15, Q.20)

Unit-II

2. (a) What is syntax analysis? What are its primary functions?

(See Unit-II, Page 43, Q.1)

(b) How do you classify the different parsing techniques?

(See Unit-II, Page 46, Q.6)

(c) How does an operator precedence parser work? Use a preconstructed operator precedence table to guide the parsing of an input 'a + b - 20' using operator precedence parser.

(See Unit-II, Page 74, Q.27)

(d) What is a translation scheme? How is it different from a syntax directed definition? Illustrate the order of execution of semantic actions in a translation scheme.

(See Unit-II, Page 115, Q.48)

Or

Describe the SLR(1) method of constructing the LR parsing table from a given grammar. Illustrate with an example.

(See Unit-II, Page 81, Q.35)

Unit-III

3. (a) What is symbol table ? How is it used in compilers ?
(See Unit-III, Page 159, Q.35)
- (b) What is an activation record ? With the help of diagram show the important fields in an activation record. (See Unit-III, Page 143, Q.20)
- (c) What are the procedure calling and returning sequences ? Explain the sequence of actions in each of them. (See Unit-IV, Page 191, Q.25)
- (d) What is run time environment ? What are the important elements of runtime environment ? How is it controlled in a program that is compiled ?
(See Unit-III, Page 140, Q.16)

Or

Explain various storage allocation strategies.

(See Unit-III, Page 145, Q.22)

Unit-IV

4. (a) Describe the backpatching technique. (See Unit-IV, Page 189, Q.22)
- (b) Draw DAG for the given block.
- $$\begin{aligned} a &= b + c \\ b &= a - d \\ c &= b + c \\ d &= a - d \end{aligned}$$
- (See Unit-IV, Page 227, Prob.17)
- (c) Translate the following expression to quadruple and triple -
 $-(X - Y)/(Z * C) - (X + Y - Z)$
(See Unit-IV, Page 201, Prob.14)
- (d) Explain briefly -
- Peephole optimization (See Unit-IV, Page 221, Q.46)
 - Register allocation (See Unit-IV, Page 207, Q.30)
 - Three address code. (See Unit-IV, Page 169, Q.2)

Or

Write down the three address code for the following switch statement -
Switch (i + j)

```
{
    case 1 : x := y - z
    case 2 : a := b + c
    case 3 : i = j + k
}
```

(See Unit-IV, Page 194, Prob.5)

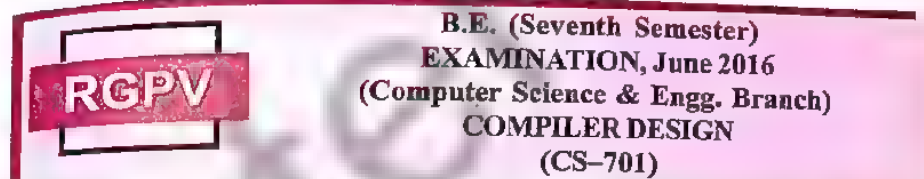
Unit-V

5. (a) What is loop optimization ? (See Unit-V, Page 236, Q.10)
- (b) Explain any two data flow properties used by target code generator for producing efficient code. (See Unit-V, Page 246, Q.27)
- (c) What is an iterative approach to solving the data flow equations ? When do we need it ? Give an example in context of computing available expression data flow property.

- (d) What are induction variables ? How does the strength reduction on induction variable help in improving loop optimization ? Illustrate with an example.
(See Unit-V, Page 251, Q.36)

Or

What is global data flow analysis ? What is its use in code optimization ?
(See Unit-V, Page 243, Q.25)



- Note : (i) Answer five questions. In each question part A, B, C is compulsory and D part has internal choice.
- (ii) All parts of each questions are to be attempted at one place.
- (iii) All questions carry equal marks, out of which part A and B (Max. 50 words) carry 2 marks, part C (Max. 100 words) carry 3 marks, part D (Max. 400 words) carry 7 marks.
- (iv) Except numericals, Derivation, Design and Drawing etc.

Unit-I

1. (a) What do you mean by Buffering ? (See Unit-I, Page 25, Q.33)
- (b) How syntax analyzer generates token ? (See Unit-I, Page 20, Q.23)
- (c) Define pass of compiler. What are the factors that decide number of passes for a compiler ? (See Unit-I, Page 6, Q.9)
- (d) Write a Lex program to find out total number of vowels and consonants from the given input string. (See Unit-I, Page 38, Prob.4)

Or

Explain various phases of compiler with the help of diagram.

(See Unit-I, Page 15, Q.20)

Unit-II

2. (a) Give the formal definition of CFG with the help of example.
(See Unit-II, Page 43, Q.2)
- (b) What do you mean by syntax tree ? (See Unit-II, Page 116, Q.52)
- (c) What are the limitations of operator precedence parsing ?
(See Unit-II, Page 75, Q.29)
- (d) Differentiate between inherited attribute and synthesized attribute with example.
(See Unit-II, Page 115, Q.50)

Or

Construct operator precedence parser.

$S \rightarrow a / \wedge / (T)$

$T \rightarrow T, S/S$

(See Unit-II, Page 80, Prob.20)

Unit-III

3. (a) What are the limitations of static allocation ?
(See Unit-III, Page 150, Q.24)
- (b) Differentiate between stack allocation and heap allocation.
(See Unit-III, Page 150, Q.26)
- (c) Write a short note on activation records. (See Unit-III, Page 143, Q.20)
- (d) By taking the example of factorial program explain how activation record will look for every recursive call in case of factorial.

Or

What do you mean by overloading of functions and operations ?

(See Unit-III, Page 137, Q.12)

Unit-IV

4. (a) What do you mean by Backpatching ? (See Unit-IV, Page 189, Q.22)
- (b) What do you mean by DAG ? (See Unit-IV, Page 217, Q.38)
- (c) Discuss some commonly used issue in code generation.
(See Unit-IV, Page 205, Q.29)
- (d) Write Quadruples and Triple for given expression.
 $Z = a - b * c \uparrow d / e + f$

(See Unit-IV, Page 197, Prob.9)

Or

Construct DAG for given expression $(X + 5) * (X + 5 + Y)$.

(See Unit-IV, Page 229, Prob.21)

Unit-V

5. (a) Define "basic blocks", flow graph with help of example.
(See Unit-IV, Page 213, Q.35)
- (b) Write short note on principle sources of optimization.
(See Unit-V, Page 233, Q.4)
- (c) What do you mean by local optimization and global optimization ?
(See Unit-V, Page 238, Q.14)
- (d) Describe the terms common sub-expression elimination and dead code elimination in the context of code optimization.

(See Unit-V, Page 236, Q.9)

Or

What do you mean by symbolic debugging of optimized code ?

(See Unit-V, Page 255, Q.42)

RGPV

B.E. (Seventh Semester)
EXAMINATION, Dec. 2016
(Computer Science & Engg. Branch)
COMPILER DESIGN
(CS-701)

Note : (i) Attempt and five questions.

(16)

(ii) All questions carry equal marks.

1. (a) What are the different phases of compiler ? Explain them with help of suitable example.
(See Unit-I, Page 15, Q.20) 7
- (b) Explain the following term in brief :
(i) Input buffering (See Unit-I, Page 25, Q.33)
(ii) Functions of lexical analyzer. (See Unit-I, Page 22, Q.26)
2. (a) Explain the concept of bootstrapping and porting in relation to compilation.
(See Unit-I, Page 10, Q.14) 7
- (b) Show whether the following grammar is LL(1) or not
 $E \rightarrow TE/+TE/\epsilon$
 $T \rightarrow FT/*FT/\epsilon$
 $F \rightarrow (E)/id$
And explain the model of predictive parser. 7
(See Unit-II, Page 59, Q.17)
3. (a) Define: Left recursive. State the rule to remove left recursive from the grammar. Eliminate left recursive from following grammar. 7
 $S \rightarrow Aa/b$
 $A \rightarrow Ac/Sd/f$
(See Unit-II, Page 56, Q.14)
- (b) Explain operator precedence parsing method with example. 7
(See Unit-II, Page 72, Q.25)
4. (a) Explain the various strategies of symbol table creation and organization.
(See Unit-III, Page 161, Q.36) 7
- (b) Define syntax directed definition. (See Unit-II, Page 110, Q.44)
Explain the various forms of syntax directed definition. 7
5. (a) Explain different storage allocation strategies with the help of suitable examples.
(See Unit-III, Page 145, Q.22) 7
- (b) Write short notes :
(i) Type checking (See Unit-III, Page 129, Q.1)
(ii) Register allocation and assignment (See Unit-IV, Page 213, Q.37)
6. (a) Construct a DAG for the basic block whose code is given below : 7
 $D := B * C$
 $E := A + B$
 $B := B * C$
 $A = E - D$
(See Unit-IV, Page 227, Prob.18)
- (b) Explain quadruple, triple and indirect triple with suitable example. 7
(See Unit-IV, Page 172, Q.4)
7. (a) Design algorithm for global common subexpression elimination. 7
(See Unit-V, Page 246, Q.28)
- (b) Explain various issues in design of code generator. 7
(See Unit-IV, Page 205, Q.29)
8. Write short notes (any four) – 14

(17)

- (a) Peephole optimization
- (b) Backpatching
- (c) Loop optimization
- (d) Dead code elimination
- (e) Syntax trees.

(See Unit-IV, Page 221, Q.46)
 (See Unit-IV, Page 189, Q.22)
 (See Unit-V, Page 236, Q.10)
 (See Unit-V, Page 236, Q.7)
 (See Unit-II, Page 116, Q.52)

RGPV

**B.E. (Seventh Semester)
 EXAMINATION, June 2017
 COMPILER DESIGN
 (CS-701)**

Note : (i) Attempt any five questions. (ii) All questions carry equal marks.

1. Write a Lex Program to convert, infix to postfix.
 (See Unit-II, Page 38, Prob.3)
2. What do you mean by Bootstrapping of compiler ?
 (See Unit-I, Page 7, Q.10)
3. Construct Top Down Parser for the Grammar
 $S \rightarrow AaAb / BbBa$
 $A \rightarrow \epsilon$
 $B \rightarrow \epsilon$
 (See Unit-II, Page 69, Prob.14)
4. Prove that Grammar is CLR but not LALR
 $S \rightarrow Aa / bAc / Bc / bBa$
 $A \rightarrow d$
 $B \rightarrow d$
 (See Unit-II, Page 100, Prob.25)
5. What are the language facilities for dynamic storage allocation ?
 (See Unit-III Page 155, Q.31)
6. Write different storage allocation strategies. (See Unit-III, Page 145, Q.22)
7. Construct DAG for the following basic block
 $d := b + c$
 $e := a + b$
 $b := b * c$
 $a := e - d$
 (See Unit-IV, Page 228, Prob.20)
8. Answer any four of the following :
 (a) Write the difference between compiler and interpreter.
 (See Unit-I, Page 12, Q.16)

- (b) Write steps of Brute force approach. (See Unit-II, Page 47, Q.7)
- (c) Write a short note on polymorphic functions.
 (See Unit-III, Page 138, Q.13)
- (d) What do you mean by Peephole optimization ?
 (See Unit-IV, Page 221, Q.46)
- (e) Compare local optimization with global optimization.
 (See Unit-V, Page 238, Q.14)
- (f) What are the properties of optimizing the compilers ?
 (See Unit-V, Page 240, Q.19)

RGPV

**B.E. (Seventh Semester) EXAMINATION, Dec. 2017
 (Computer Science & Engg. Branch)
 Grading System (GS)
 COMPILER DESIGN
 CS-701 (GS)**

Note : (i) Attempt any five questions.

(ii) All questions carry equal marks.

1. (a) What is meant by Input buffering ? Explain the use of sentinels in recognizing tokens.
 (See Unit-I, Page 25, Q.33) 7
 (b) Explain the various phases of compiler with the help of diagram. Take any one example to elaborate complete working of compiler.
 (See Unit-I, Page 15, Q.20) 7
2. (a) Write a lex specification and the pattern matching routine to display the identifier and the line number of its occurrences. 7
 (b) Explain left recursion and show how it is eliminated. Describe the algorithm used for eliminating left recursion. (See Unit-II, Page 56, Q.13) 7
3. (a) Show that the following grammar is LR(1) but not LALR(1). 7
 $S \rightarrow Aa/bAc/Bc/bBa$
 $A \rightarrow d$
 $B \rightarrow d$
 (See Unit-II, Page 100, Prob.25)
 (b) What are the causes of backtracking in top down parser ? Explain with an example.
 (See Unit-II, Page 49, Q.9) 7
4. (a) Assuming suitable syntax directed definition, construct a syntax tree for the expression $a - 4 + e$.
 (See Unit-II, Page 113, Q.45) 7
 (b) Consider the grammar -
 $S \rightarrow ACB/CbB/Ba$
 $A \rightarrow da/BC$
 $B \rightarrow g/\epsilon$
 $C \rightarrow h/\epsilon$
 Calculate FIRST and FOLLOW.
 (See Unit-II, Page 66, Prob.9)

5. (a) What are the difficulties faced by memory allocation for variable length requirements? Under what circumstances does external fragmentation happen. (See Unit-III, Page 159, Q.34) 7
(b) Explain the concept of type checking and type conversion with an example. (See Unit-III, Page 136, Q.10) 7
6. (a) Translate the following code into a three address code – 7
 $a = 10;$
 $b = 6;$
 if ($a > b + 5$)
 $b = 5;$ (See Unit-IV, Page 195, Prob.6)
 (b) What is DAG? Construct DAG for the basic block – 7
 $D := B * C$
 $E := A + B$
 $B := B * C$
 $A := E - D$ (See Unit-IV, Page 217, Q.39)
7. (a) Analyse the possible causes of a dead code. Explain with an example how the compiler can catch the presence of a dead code. 7
 (See Unit-V, Page 236, Q.7)
 (b) Explain the common subexpression elimination copy propagation and transformation for moving loop invariant computations in detail. 7
 (See Unit-V, Page 249, Q.33)
8. Write short notes on – 14
 (a) Peephole optimization (See Unit-IV, Page 221, Q.46)
 (b) Symbol table (See Unit-III, Page 159, Q.35)
 (c) Global data flow analysis (See Unit-V, Page 243, Q.25)
 (d) Backpatching. (See Unit-IV, Page 189, Q.20)

RGPV

CS-701 (GS)
B.E. VII Semester
EXAMINATION, May 2018
Grading System (GS)
COMPILER DESIGN

Note: (i) Attempt any five questions.
 (ii) All questions carry equal marks.

1. (a) What is front end and back end of compiler? What are the advantages of breaking up the compiler functionality into these two stages? 7
 (See Unit-I, Page 12, Q.17)
 (b) What are the various components of lexical specification file?

2. (a) Illustrate with an example. (See Unit-I, Page 35, Q.40) 7
 What are the main design issues to be considered while generating code for lexical analyzer given the minimized DFA? 7
 (See Unit-IV, Page 208, Q.31)
 (b) What is a context-free grammar? Illustrate with an example the different components of a context-free grammar. What are the advantages of using CFG to specify a language? 7
 (See Unit-II, Page 45, Q.4)
3. (a) Illustrate the steps in the parsing of an input ' $x = y + z - 5$;' by an LR parser using a predictive constructed LR passing table. 7
 (b) Distinguish between top-down parsing and bottom-up parsing. What is the largest class of grammars that can be parsed by each of them? (See Unit-II, Page 63, Q.23) 7
4. (a) What is a symbol table? Explain how the symbol table in a compiler can be implemented by a hash table. (See Unit-III, Page 163, Q.37) 7
 (b) Explain briefly – 7
 (i) Type conversion (See Unit-III, Page 135, Q.9)
 (ii) Dynamic storage allocation. (See Unit-III, Page 157, Q.33)
5. (a) Explain static and dynamic type checking with example. 7
 (See Unit-III, Page 135, Q.8)
 (b) How can three address code be implemented in a compiler? Describe triples and indirect triples method of implementing TAC with example. (See Unit-IV, Page 172, Q.5) 7
6. (a) Define backpatching and semantic rules for boolean expression. How DAG is different from syntax tree? (See Unit-IV, Page 190, Q.23) 7
 (b) Construct the DAG for the following basic block – 7
 $a := b + c$
 $b := b - d$
 $c := c + d$
 $e := b + c$
 (See Unit-IV, Page 230, Prob.25)
7. (a) Explain following code improving transformations with examples – 7
 (i) Local and global elimination of common sub expression
 (ii) Copy propagation and dead code elimination. (See Unit-V, Page 247, Q.29)
 (b) Give a brief on peephole optimization. (See Unit-IV, Page 221, Q.46)

8. Write short notes –
 (a) Optimization of basic block
 (b) Induction variable elimination
 (c) Local and loop optimization.

(See Unit-V, Page 240, Q.20)
 (See Unit-V, Page 252, Q.37)
 (See Unit-V, Page 239, Q.15)

RGPV

CS-7002 (CBGS)
B.E. VII Semester
EXAMINATION, November 2018
Choice Based Grading System (CBGS)
COMPIER DESIGN

Note : (i) Attempt any five questions.
 (ii) All questions carry equal marks.

1. (a) What are the phases of a compiler ? Explain the function of each phase in brief with examples. (See Unit-I, Page 15, Q.20) 7
 (b) Explain the following terms –
 (i) Translators, compiler and interpreters
 (ii) Bootstrapping. (See Unit-I, Page 15, Q.19)
2. (a) What are major data structures used in compiler ? Explain the concept of bootstrapping. (See Unit-I, Page 8, Q.11) 7
 (b) What do you mean by context-free grammar ? Give distinction between regular and context-free grammar and limitations of context-free grammar. (See Unit-II, Page 45, Q.3) 7
3. Consider the following grammar –
 $E \rightarrow E + T/T$
 $T \rightarrow TF/F$
 $F \rightarrow F*/a/b$
 (i) Construct the SLR parsing table for this grammar
 (ii) Construct the LALR parsing table. (See Unit-II, Page 106, Prob.27) 14
4. (a) Write down the procedure to translate an infix expression into postfix form. Also write syntax directed definition for the same. (See Unit-II, Page 127, Q.62) 7
 (b) Explain the symbol table management system. (See Unit-III, Page 159, Q.35) 7
5. (a) Write a short note on operator precedence parsing and function. (See Unit-II, Page 73, Q.20) 7
 (b) Differentiate between stack allocation and heap allocation. (See Unit-III, Page 150, Q.20) 7

6. (a) Construct a DAG for the basic block whose code is given below – 7
 $D := B * C$
 $E := A + B$
 $B := B * C$
 $A := E - D$

(See Unit-IV, Page 227, Prob.18)

- (b) What is peephole optimization ? Explain it. 7
 (See Unit-IV, Page 221, Q.46)
7. (a) Explain in brief the various issues of design of a code generator. 7
 (See Unit-IV, Page 205, Q.29)
 (b) Explain the basic block and control flow graph. 7
 (See Unit-IV, Page 213, Q.35)
8. Write short notes – 14
 (i) Three address code (See Unit-IV, Page 169, Q.2)
 (ii) Loop optimization (See Unit-V, Page 236, Q.10)
 (iii) Dead code elimination (See Unit-V, Page 236, Q.7)
 (iv) Backpatching. (See Unit-IV, Page 189, Q.22)

RGPV

CS-7002 (CBGS)
B.E. VII Semester
EXAMINATION, May 2019
Choice Based Grading System (CBGS)
COMPIER DESIGN

Note : (i) Attempt any five questions.
 (ii) All questions carry equal marks.

1. (a) What is a front and back end of a compiler ? What are the advantages of braking up the compiler functionality into these two distinct stages ? (See Unit-I, Page 12, Q.17)
 (b) Why the analysis portion of a compiler is normally separated into lexical analysis and parsing phasor ? Explain tokens, patterns and lexemes with an example. (See Unit-I, Page 24, Q.31)
2. (a) What is a regular definition ? Specify an identifier in C language taking the help of regular definitions. Explain butter pairs and sentinels. (See Unit-I, Page 36, Q.42)
 (b) How do you classify the different parsing techniques ? Define left recursion and left factoring. Consider the following grammar –
 $A \rightarrow ABd|Aa|a$
 $B \rightarrow Be|b$
 remove left recursion.

(See Unit-II, Page 56, Q.12)

3. (a) What is a recursive-descent parsing ? Define FIRST and FOLLOW functions. Consider the grammar –
 $S \rightarrow ACB \mid Cb B \mid Ba$
 $A \rightarrow da \mid BC$
 $B \rightarrow g \mid \epsilon$
 $C \rightarrow h \mid \epsilon$
Calculate FIRST and FOLLOW.
(See Unit-II, Page 66, Prob.10)
- (b) What are the components of a table-driven predictive parser ? Write table-driven predictive parsing algorithm.
(See Unit-II, Page 59, Q.16)
4. (a) Discuss error recovery in predictive parsing. Explain shift reduce parsing with taking suitable examples. (See Unit-II, Page 62, Q.22)
- (b) What is a syntax-directed definition ? What are its main characteristics ? Illustrate with an example. Explain S-attribute and L-attribute definition.
(See Unit-II, Page 121, Q.57)
5. (a) Explain activation trees and activation records with an suitable examples. Also explain calling sequence.
(See Unit-III, Page 149, Q.23)
- (b) Discuss heap management and garbage collection with an example.
(See Unit-III, Page 157, Q.32)
6. (a) What is an intermediate code ? What are the benefits of intermediate code generation ? How can three address code be implemented in a compiler ? Describe quadruple. (See Unit-IV, Page 172, Q.6)
- (b) Define backpatching and semantic rules for Boolean expression.
(See Unit-IV, Page 191, Q.24)
- Derive the three address code for the following expression $P < Q$ and $R < S$ and $T < U$.
7. (a) Discuss the issues in the design of code generator. Also discuss the DAG representation of basic blocks. (See Unit-IV, Page 220, Q.44)
- (b) How is a call to a procedure translated into TAC ? Illustrate with an example.
(See Unit-IV, Page 192, Q.26)
8. (a) What is optimization ? What are the common techniques for improving the intermediate code. Explain three of them in brief.
(See Unit-V, Page 248, Q.31)
- (b) What are dead code elimination and strength reduction transformation ? Illustrate with an example.
(See Unit-V, Page 236, Q.6)